Assignment 4 - LaTeX Write-Up

Connor Fleischman

December 6, 2024

Contents

1 Introduction

Assignment 4 requires implementing a Directed Graph (2) and some Spices and Knapsacks (3) and running algorithms on them. To do this I modified the Undirected Graph class from **Assignment 3** to include weighted edges between the vertices. Then, with each graph described in `graphs2.txt` (2.1) , beginning with the first vertex in the graph, a Single Source Shortest Path algorithm is performed (2.2) . Next, after building the spices and knapsacks described in `spice.txt` (3.1) , each knapsack is filled using the Fractional Knapsack greedy algorithm (3.2). Finally, all graphs, spices, and knapsacks are destroyed and their memory freed (4).

2 Directed Graphing

To implement a Directed Graph and perform a Single Source Shortest Path (SSSP) algorithm on the first vertex, one must:

- Read and parse the data in `graph2.txt`
- Build each graph described in the file
- Perform a SSSP algorithm on the first vertex to all other connected vertices
- Destroy the graph and allocate memory

2.1 Parsing & Building

The `graph2.txt` file is formatted in such a way so that each line contains one instruction. Each graph begins with the 'new' command followed by some number of 'new vertex' or 'new edge' commands. Lines beginning with '-', the comment symbol, are dropped.

Once parsing is complete this vector of vectors of strings, graphs, is built. In the main file, each individual graph in graphs is put through the build function. Build takes every instruction in a graph and builds that graph with some specified number of vertices and edges (with their own weights).

2.1.1 Implementation

```
1  std::vector<std::vector<std::string>> parseGraph() // Returns a vector of graphs, consisting of a vector of strings,
       where each string is a vertex or edge, from the file: graphs2.txt
2  {
3     std::vector<std::vector<std::string>> graphs; // Declare graphs as a vector of vectors of strings
4     std::ifstream file("./input/graphs2.txt");    // Declare file and open file at location
5
6     if (file.is_open()) // If the file is open
7     {
8        std::vector<std::string> currentGraph; // Declare currentGraph as a vector of strings
9        std::string line;                      // Declare line as a string
10       while (std::getline(file, line))       // For every line in the file
11       {
12          if (line.empty() || line.front() == '-') // If the line is empty or a comment line
13          {
14             continue; // Skip this line
15          }
16          if (line == "new graph") // If the line declares a new graph
17          {
18             if (!currentGraph.empty()) // If the current graph storage is not empty
19             {
20                graphs.push_back(currentGraph); // Push the current graph back to the graphs vector
21                currentGraph.clear();           // Clear the current graph vector for the next graph to be read
22             }
```

```
            continue; // Skip this line
        }
        // Non-skiped lines:
        currentGraph.push_back(line); // Push the line to the current graph
        // Continue to the next line
    }
    // When all lines in the file are read
    if (!currentGraph.empty()) // If there is still a graph in the current graph vector
    {
        graphs.push_back(currentGraph); // Push this final graph back to the vector of graphs
    }

    file.close(); // Close the file
}
else // If the file is not open
{
    std::cout << "Error opening file" << std::endl;
}
return graphs; // Return the vector of vectors of strings
}
```

Listing 1: Parsing Implementation

My implementation opens and reads the file. Then every line in the file is read and parsed. Comments are skipped and all other lines are pushed back to the returned graphs vector of vectors. Every 'new graph' command starts a new graph vector which, when finished being parsed, will be pushed back into the larger vector.

```
std::string buildGraph(std::vector<std::string> &graphInstructions, Graph &graph) // Given the instructions for one
    graph and the graph, build that graph
{
    bool foundFirstVertex = false;                    // Flag to track if first vertex is found
    std::string firstVertexID;                        // Variable to hold first vertex ID
    for (std::string &instruction : graphInstructions) // For every instruction in the vector of instructions
    {
        std::istringstream inst(instruction);    // Break the instruction into individual words
        std::string identifier, opcode, operand; // Declare both identifier, opcode, and operand as strings
        inst >> identifier >> opcode;            // Read the first two words to their strings
        std::getline(inst, operand);             // Read the remaining words to the string

        if (opcode == "vertex") // If the opcode is named vertex
        {
            if (!foundFirstVertex) // If the first vertex is not recorded
            {
                std::istringstream vID(operand); // Break the operand into just the string, no white-space
                std::string num;                 // Declare num as a string
                vID >> num;                      // Read the vertex ID from the operand into num
                firstVertexID = num;             // Record the first vertex
                foundFirstVertex = true;         // Set the flag
                insertVertex(num, graph);        // Add a vertex with id: num to the graph
            }
            else
            {
                std::istringstream vID(operand); // Break the operand into just the string, no white-space
                std::string num;                 // Declare num as a string
                vID >> num;                      // Read the vertex ID from the operand into num
                insertVertex(num, graph);        // Add a vertex with id: num to the graph
            }
        }
        else if (opcode == "edge") // If the opcode is edge
        {
            std::istringstream edge(operand);     // Break the operand into individual words
            std::string v1, bridge, v2;           // Declare the vertices being connected and the connection symbol
            int weight;                           // Delcare the weight of the connection
            edge >> v1 >> bridge >> v2 >> weight; // Read the first three words and the weight to these strings and int
            insertEdge(v1, v2, weight, graph);    // Add a edge from v1 - v2 with some weight
        }
    }
    foundFirstVertex = false; // Reset the flag
    return firstVertexID;
}
```

Listing 2: Building Implementation

Firstly, a flag to track the first vertex in the graph and a placeholder for the first vertex's ID are created. Then every instruction in the provided graph is read and interpreted. The instruction is broken into individual words. Next, depending on the first word, an edge or vertex is built for that graph. After all instructions are interpreted, the first vertex's ID is returned.

2.1.2 Results

```
Building graph #1
Vertex #1 created
Vertex #2 created
Vertex #3 created
Vertex #4 created
Vertex #5 created
Edge created from vertex #2 to #3 with weight: 5
Edge created from vertex #2 to #4 with weight: 8
Edge created from vertex #2 to #5 with weight: -4
Edge created from vertex #3 to #2 with weight: -2
Edge created from vertex #4 to #3 with weight: -3
Edge created from vertex #4 to #5 with weight: 9
Edge created from vertex #5 to #3 with weight: 7
Edge created from vertex #5 to #1 with weight: 2
Edge created from vertex #1 to #2 with weight: 6
Edge created from vertex #1 to #4 with weight: 7
```

Here we see that, for **Graph 1**, five vertices are created and ten edges are created between the vertices. Each characterized by some weight.

2.2 Single Source Shortest Path Algorithm

After the graph is built, using the first vertex returned by the build function, a Single Source Shortest Path algorithm is ran. This algorithm calculates the path of least cost from some source to every available sink. It does this by performing every possible traversal from the source to every possible sink. While doing this it keeps track of the most efficient, lest costly, path. Once all pathways are traversed, and the best routes calculated, the program displays these shortest paths.

2.2.1 Implementation

```
1   bool mapPathways(std::string &startID) // Constructs the pathways between vertices from a starting vertex
2   {
3      Vertex *startVertex = search(startID); // Search for the starting vertex using it's ID
4      if (!startVertex)                     // If the starting vertex is not found
5      {
6         std::cout << "Vertex #" << startID << " not found" << std::endl;
7         return false;
8      }
9      // If the starting vertex is found
10     startVertex->distance = 0;                    // Set the starting vertex's distance to 0
11     for (int i = 0; i < vertices.size() - 1; i++) // For every item in the list of vertices - 1
12     {
13        for (Edge *edge : edges) // For every edge in edges
14        {
15           Vertex *startEdgeVertex = search(edge->startID); // Set the starting edge vertex to that edge's start
      vertex
16           Vertex *endEdgeVertex = search(edge->endID);     // Set the ending edge vertex to that edge's end vertex
17
18           if (endEdgeVertex->distance > startEdgeVertex->distance + edge->weight) // Relax function to check if a
      shorter path exists
19           {
20              endEdgeVertex->distance = startEdgeVertex->distance + edge->weight; // Setting the shortest path
21              endEdgeVertex->predecessor = startEdgeVertex;                       // Setting the predecessor vertex
22           }
23        }
24     }
25     // After all vertices have been mapped out
26     for (Edge *edge : edges) // For every edge in edges
27     {
28        Vertex *startEdgeVertex = search(edge->startID);                      // Set the starting edge vertex to
      that edge's start vertex
29        Vertex *endEdgeVertex = search(edge->endID);                          // Set the ending edge vertex to that
       edge's end vertex
```

```
30          if (endEdgeVertex->distance > startEdgeVertex->distance + edge->weight) // If a shorter path is found after,
        then there is a negative weight cycle
31          {
32              std::cout << "<< Graph contains a negative-weight cycle >>" << std::endl;
33              return false;
34          }
35       }
36       // If no shorter paths exist
37       for (Vertex *vertex : vertices)
38       {
39          if (vertex->id != startID) // For all vertices not the start
40          {
41              std::cout << "#" << startID << std::setw(3) << " -> #" << vertex->id << " | Cost: " << std::setw(3) <<
        vertex->distance << " | Path: ";
42              printPath(vertex->id); // Prints the path from the source to this vertex
43              std::cout << std::endl;
44          }
45       }
46       return true;
47    }
48 };
```

Listing 3: SSSP Implementation

In the above code we see that mapPathways takes in some starting vertex ID and searches for the vertex. Once found, its distance is set to 0, since it's the source. Then for every other vertex, the path from the source to that vertex is calculated. If that path is shorter than the existing path to that vertex, it is updated. Then once all vertices have been mapped a check is ran to see if all shortest paths were mapped. Finally the result is output along with the path taken to get from the source to that sink.

The below function is a helper-function for outputting the shortest path:

```
1    void printPath(std::string &vertexID) // Private function to print the path from some source to some sink
2    {
3       Vertex *currentVertex = search(vertexID);  // Find the current vertex
4       if (currentVertex->predecessor != nullptr) // If the current vertex exists
5       {
6          printPath(currentVertex->predecessor->id); // Recurse on that vertex's predecessor until no more predecessors
        are found
7          std::cout << " -> ";
8       }
9       std::cout << currentVertex->id; // Print the vertex
10   }
```

Listing 4: SSSP Helper

This uses recursion to take the path of the most efficient route from sink to source and reorder it in reverse to get the correct path displayed. Resulting in a path from source to sink, instead of backwards.

2.2.2 Results

```
Performing a Single Source, Shortest Path algorithm begining at vertex #1
#1 -> #2 | Cost:   2 | Path: 1 -> 4 -> 3 -> 2
#1 -> #3 | Cost:   4 | Path: 1 -> 4 -> 3
#1 -> #4 | Cost:   7 | Path: 1 -> 4
#1 -> #5 | Cost:  -2 | Path: 1 -> 4 -> 3 -> 2 -> 5
All pathways from vertex #1 to other vertices traversed
```

Here we see that the SSSP was successful in calculating the shortest path, path of least cost, between the first vertex, **#1**, to the other four vertices. Lastly, after comparing my results to those provided for **Graph 1**, no errors were found, meaning the algorithm ran properly.

3 Spices & Knapsacks

The remaining workload of **Assignment 4** consists of creating certain spices and knapsacks. Each spice is characteristics by a color, some total price, and a quantity. The unit price for a spice is also calculated. A knapsack is characterized simply as some number, describing the capacity of the knapsack.

To fulfill this we must:

- Read and parse the spices and knapsacks in `spices.txt`
- Build all spices with their respective characteristics
- Build all knapsacks with their respective capacities
- Perform a fractional knapsack algorithm for each knapsack on all spices
- Destroy all knapsacks and spices and allocate memory

3.1 Parsing & Building

The input file `spices.txt`, being so similarly formatted to `graph2.txt`, requires some of the same logic as before too. However, unlike before, `spices.txt` does not have one instruction per line. Instead, the line containing the spice characteristics contains multiple instruction, one per characteristic. This results in an increase in complexity as when parsing a spice, we must parse each instruction on the line, instead of just the whole line.

To achieve this, I broke my logic into two main sections, parsing the whole of `spice.txt`, and parsing a specific line into its individual instructions. This allows for reduced complexity and an easier overall time understanding the code.

3.1.1 Implementation

```
1  std::vector<std::string> parseInstructions() // Parse the data from spice.txt into a vector of strings
2  {
3     std::vector<std::string> instructions;   // Declare vector of strings to hold instructions
4     std::ifstream file("./input/spice.txt"); // Read and open the file
5
6     if (file.is_open()) // If the file is open
7     {
8        std::string line;                // Placeholder for current line
9        while (std::getline(file, line)) // While there is a line in the file
10       {
11          if (line.empty() || line.front() == '-') // If the line is empty or a comment
12          {
13             continue; // Skip
14          }
15          // For remaining lines:
16          instructions.push_back(line);
17       }
18    }
19    else // If the file is not open
20    {
21       std::cout << "Error opening file" << std::endl;
22    }
23    return instructions;
24 }
```

Listing 5: Parsing File Implementation

Above consists of half of the parsing for the spices and knapsacks. Again, the file is open and read, each line is parsed, comments skipped, and the vector of instructions is returned. Breaking the file down in this way allowed me to make parsing the characteristics more dynamic as well.

```cpp
std::vector<std::string> parseLine(std::string &line) // Parse the instruction into its subinstructions
{
   std::vector<std::string> instructions; // List of instructions in this line
   std::string command;                   // Placeholder for current command
   bool equalSign = false;                // Flag to track if passed '='

   for (char c : line) // For every character in the instruction
   {
      if (c == ';') // If the end of the command is reached
      {
         instructions.push_back(command); // Push the command back to the list
         command.clear();                 // Clear the command placeholder
         equalSign = false;               // Reset the flag
      }
      else // If the end of the command has not ended
      {
         if (!equalSign) // If the flag is not set
         {
            if (c == '=') // If the current char is '='
            {
               equalSign = true; // Set the flag
            }
         }
         else // If the flag is set
         {
            if (!isspace(c)) // If the char is not a space
            {
               command += c; // Add it to the command
            }
         }
      }
   }
   // After every command in the instruction has been read
   return instructions;
}
```

Listing 6: Parsing Instruction Implementation

The parseLine function handles breaking down an instruction into its commands. It does that by taking a line and breaking it into chunks separated by ';'. Then, since each chunk consists of the type and the value (separated by '='), only the data after '=' is kept. This is done by reading each character in the line, and once it reaches a '=' it pushes the characters back to a string until stopping at ';'. Then it is pushed back to the vector of strings. After each subcommand in the instruction is separated, it returns this vector of commands.

Below we have the build functions which constructs the spices, with their defined characteristics, and the knapsacks, with their specified size. The buildSpices function takes in all the instructions and the container for all the spices. Then, using only instructions beginning with 'spice', it breaks the spice command into its subcommands and adds a spice, created within addSpice, to the spices container.

```cpp
void buildSpices(std::vector<std::string> &instructions, Spices &spices)
{
   for (std::string &instruction : instructions) // For every instruction in the list
   {
      std::istringstream inst(instruction); // Break that instruction into words
      std::string identifier, excess;       // Declare identifier and excess as strings
      inst >> identifier;                   // Take the identifier off the instruction
      std::getline(inst, excess);           // Read the remaining words to excess

      if (identifier == "spice") // If the identifier is spice
      {
         std::vector<std::string> commands = parseLine(excess);                        // Break the excess into its commands
         addSpice(commands[0], std::stoi(commands[1]), std::stoi(commands[2]), spices); // Build the spice using each command, parsed to an int if necessary
      }
   }
   std::cout << "Spices built" << std::endl;
}
```

Listing 7: Building Spices Implementation

Next the knapsacks are built. Just like before, all instructions and the knapsack container are passed in. The instructions starting with 'knapsack' are broke down to just

its data after the '=', using parseLine. Then the knapsack, created within the addKnapsack function, is added to the knapsack container, converting the string command to an int for the size.

```
void buildKnapsacks(std::vector<std::string> &instructions, std::vector<Knapsack *> &knapsacks)
{
   for (std::string &instruction : instructions) // For every instruction in the list
   {
      std::istringstream inst(instruction); // Break that instruction into words
      std::string identifier, excess;       // Declare identifier and excess as strings
      inst >> identifier;                   // Take the identifier off the instruction
      std::getline(inst, excess);           // Read the remaining words to excess

      if (identifier == "knapsack") // If the identifier is knapsack
      {
         std::vector<std::string> commands = parseLine(excess); // Break the excess into its commands
         addKnapsack(std::stoi(commands[0]), knapsacks);        // Build the knapsack using the command parsed to an
    int
      }
   }
   std::cout << "Knapsacks built" << std::endl;
}
```

Listing 8: Building Knapsacks Implementation

3.1.2 Results

As we can see in the image below, all four spices from `spice.txt` were parsed and built correctly, with each spice's characteristics displayed too. Also, all five knapsacks were constructed with their respective size as an integer, parsed and built from strings in the file.

```
Building Spices and Knapsacks
[Constructed] Spice color:     red | totalPrice:   4 | quantity:   4 | unitPrice:   1
[Constructed] Spice color:   green | totalPrice:  12 | quantity:   6 | unitPrice:   2
[Constructed] Spice color:    blue | totalPrice:  40 | quantity:   8 | unitPrice:   5
[Constructed] Spice color:  orange | totalPrice:  18 | quantity:   2 | unitPrice:   9
Spices built
[Constructed] Knapsack with capacity:   1
[Constructed] Knapsack with capacity:   6
[Constructed] Knapsack with capacity:  10
[Constructed] Knapsack with capacity:  20
[Constructed] Knapsack with capacity:  21
Knapsacks built
```

3.2 Fractional Knapsack Algorithm

Now that the spices and knapsacks have been successfully parsed and built, each knapsack is to be filled with spice. This is done by using the fractional knapsack greedy algorithm. Unlike the 0-1 knapsack problem, where only whole amounts of spice are allowed, fractional knapsack allows for percentages of spice. 0-1 knapsack is not solvable using a greedy algorithm, so dynamic programming is a necessity; fractional knapsack can be solved greedily and not dynamically.

3.2.1 Implementation

Building out the logic for fractional knapsack took some time. The first step to having a working algorithm is properly ordering the spices in its container. Using the my

selection sort implementation from **Assignment 1**, with some small modifications, I reordered the spices by their unit price. Then, starting with the most valuable spice, each spice is put in the knapsack until either the spice is gone, or the knapsack is full.

```
1   void fillKnapsack(Knapsack *knapsack) // Use a greedy algorithm to fill this knapsack
2   {
3       spices = selectionSort(spices);      // Sort the spices's by their unit prices
4       double sum = 0.0;                    // Placeholder for total value of current knapsack
5       int emptySpace = knapsack->capacity; // Placeholder to calculate the remaining space in the knapsack
6       std::cout << "Filling knapsack with capacity: " << knapsack->capacity << std::endl;
7
8       for (Spice *spice : spices) // For each spice
9       {
10          if (emptySpace == 0) // If there is no space left in the knapsack
11          {
12              break;
13          }
14          if (spice->quantity <= emptySpace) // If the this spice can fit in the knapsack
15          {
16              sum += spice->totalPrice;      // Add the price per quantity to the sum
17              emptySpace -= spice->quantity; // Update the remaining space in the knapsack
18              std::cout << "Took all of the " << spice->color << " spice." << std::endl;
19          }
20          else // If the spice cannot fit in the knapsack
21          {
22              double proportion = static_cast<double>(emptySpace) / spice->quantity; // Calculate the percentage of the
    spice to be took
23              sum += spice->totalPrice * proportion;                                 // Add the price per quantity,
    based on this fractional quantity
24              std::cout << "Took " << proportion * 100 << "% of the " << spice->color << " spice." << std::endl;
25              emptySpace = 0; // Since were taking a fractional size, we will always be full after
26          }
27      }
28      // Formatting output to only display if the knapsack is not full, how much space remains
29      emptySpace == 0 ? std::cout << "Knapsack filled to capacity." << std::endl : std::cout << "Knapsack cannot be
    filled anymore, remaining space: " << emptySpace << std::endl;
30      std::cout << "Knapsack total value: " << sum << std::endl;
31      std::cout << "-------------------------------------------------" << std::endl;
32  }
33 };
```

Listing 9: Fractional Knapsack Implementation

The function fillKnapsack takes in the knapsack to be filled, fills it, and reports its contents. It does this by tracking the remaining space in the knapsack and filling the knapsack with the whole of a spice only if it will all fit. Then for the remaining space it will take a fraction of spice to fill the space. If the remaining space could not be filled by a fraction, it is also reported. This is performed on every knapsack.

3.2.2 Results

Below are two filled knapsacks. I've chosen these two knapsacks because they represent the two main challenges in this section. Firstly, the knapsack with a capacity of 1. This knapsack is too small for any one spice quantity so only a fractional quantity will fit. As we see below, when this knapsack is filled, it can only store half of the orange spice. Since orange costs 9 per quantity, and since we can only store one, the knapsack is filled and worth 9.

```
Greedily filling Knapsacks with Spices
Filling knapsack with capacity: 1
Took 50% of the orange spice.
Knapsack filled to capacity.
Knapsack total value: 9
```

Second, the knapsack with a capacity of 21. This knapsack is too big for all the spice. In total there are 20 units of spice, and since this is within the bounds of the knapsack, all spices are put in. However, there is still more room in the sack, so the remaining space is recorded along with the contents and value of the knapsack.

```
Filling knapsack with capacity: 21
Took all of the orange spice.
Took all of the blue spice.
Took all of the green spice.
Took all of the red spice.
Knapsack cannot be filled anymore, remaining space: 1
Knapsack total value: 74
```

4 Clean Up

After each graph is built and the SSSP calculated, it is automatically rewritten with the next graph. Once all graphs have been realized, the spices and knapsacks are parsed and constructed. Then the fractional knapsack greedy algorithm is performed. Finally, when everything has finished running, the destructors are called and the modified graph, the spices, and the knapsacks are destroyed.

```
-- Destroying Graph --
-- Graph Destroyed --
```

Above we see that the graph is destroyed successfully, below we see that after the knapsacks are filled, the knapsacks are destroyed, then the spices.

```
All Knapsacks filled with Spices
Knapsacks destroyed
Spices destroyed
```

These destructor calls finish my program and completely clean up the memory accessed and modified in my code.

The remaining sections of this document describe the various functions and class structures used to complete this assignment that were not explicitly mentioned prior.

5 Miscellaneous Implementations

The main file in my program consists of three functions. The main function which houses the two function calls for graphing and for the spices and knapsacks operations. And the two other functions, each consisting of the order in which parsing, building, and operating on the data is defined.

```
1  void interactGraph(Graph &graph)
2  {
3     int numGraphs = 0;                                          // Counter to track the number of graphs being
        produced
4     for (vector<string> graphInstructions : buildInstructions()) // For every graph in the blueprint returned by
        buildInstructions()
5     {
6        numGraphs++; // Increment for every graph built
7        cout << "Building graph #" << numGraphs << endl;
8        string v1 = buildGraph(graphInstructions, graph); // Build this graph getting its first vertex
9        cout << "Performing a Single Source, Shortest Path algorithm begining at vertex #" << v1 << endl;
10       mapPathways(v1, graph); // Map the pathways from v1 to all other vertices
11       cout << "All pathways from vertex #" << v1 << " to other vertices traversed" << endl;
12       cout << "Destroying graph #" << numGraphs << endl;
13       clearGraph(graph); // Destroy the graph
14       cout << "-------------------------------------------------" << endl;
15    }
16    numGraphs = 0;
17 }
```

```
18
19 void interactSpices(Spices &spices, std::vector<Knapsack *> &knapsacks)
20 {
21    std::vector<std::string> instructions = parseInstructions();
22    cout << "Building Spices and Knapsacks" << endl;
23    buildSpices(instructions, spices);       // Build all spices using the parsed instructions
24    buildKnapsacks(instructions, knapsacks); // Build all knapsacks using the parsed instructions
25    cout << "-------------------------------------------------" << endl;
26    cout << "Greedily filling Knapsacks with Spices" << endl;
27    fillKnapsacks(knapsacks, spices); // Fill knapsacks using a Greedy Algorithm and return its contents
28    cout << "All Knapsacks filled with Spices" << endl;
29    knapsacks.clear(); // Empty the list of knapsacks
30    cout << "Knapsacks destroyed" << endl;
31 }
32
33 int main()
34 {
35    Graph graph;
36    interactGraph(graph);
37    cout << "-------------------------------------------------" << endl;
38    Spices spices;
39    std::vector<Knapsack *> knapsacks;
40    interactSpices(spices, knapsacks);
41    return 0;
42 }
```

Listing 10: Main Implementation

The DirectedGraph file houses the **Graph** class. Each graph consists of a vector of vertices and a vector of edges. A vertex contains an id, a processed flag, its distance to the single source, the predecessor vertex, and all neighboring vertices.

```
1  class Vertex // Defines the Vertex class
2  {
3  public:
4     std::string id;                  // Identifies the id of the vertex
5     bool processed;                  // Identifies the processed flag for the vertex
6     int distance;                    // Identifies the distance from the source to this vertex
7     Vertex *predecessor;             // Identifies the predecessor vertex for Bellman-Ford
8     std::vector<Vertex *> neighbors; // Identifies the list of neighbors of the vertex
9
10    Vertex(std::string &idValue) // Vertex constructor
11    {
12       this->id = idValue;          // Set this vertex's id to some given value
13       this->processed = false;     // Set this vertex's processed flag to false
14       this->distance = 8675309;    // Jenny don't change your number
15       this->predecessor = nullptr; // Set this vertex's predecessor to nullptr
16    }
17 };
```

Listing 11: Vertex Implementation

An edge contains the starting and ending vertices's IDs as well as some weight or cost.

```
1  class Edge // Defines the Edge class
2  {
3  public:
4     std::string startID; // Identifies the starting vertex ID of the edge
5     std::string endID;   // Identifies the ending vertex ID of the edge
6     int weight;          // Identifies the weight of the edge
7
8     Edge(std::string &v1ID, std::string &v2ID, int cost) // Edge constructor
9     {
10       this->startID = v1ID; // Set this edge's start to v1
11       this->endID = v2ID;   // Set this edge's end to v2
12       this->weight = cost;  // Set this edge's weight to cost
13    }
14 };
```

Listing 12: Edge Implementation

The SpiceKnapsack file houses the classes for a **Spice** and a **Knapsack**. A spice is characterized by some color, the total price of the spice, and the quantity of the spice. Using the price and quantity, the total unit price can be derived.

```
class Spice // Defines the Spice class
{
public:
   std::string color; // Identifies the color of the spice
   int totalPrice;    // Identifies the total price of the spice
   int quantity;      // Identifies the quantity of the spice
   double unitPrice;  // Identifies the unit price of the spice

   Spice(std::string &spiceColor, int totalCost, int totalQuantity) // Spice constructor
   {
      this->color = spiceColor;                                     // Set this spice's color to some given color
      this->totalPrice = totalCost;                                 // Set this spice's total price to some given
     cost
      this->quantity = totalQuantity;                               // Set this spice's quantity to some given
     amount
      this->unitPrice = static_cast<double>(totalCost) / totalQuantity; // Set this spice's unit price to the average
     cost per amount of spice
      std::cout << "[Constructed] Spice color: " << std::setw(7) << this->color << " | totalPrice: " << std::setw(3)
     << this->totalPrice << " | quantity: " << std::setw(3) << this->quantity << " | unitPrice: " << std::setw(3) <<
     this->unitPrice << std::endl;
   }
};
```

Listing 13: Spice Implementation

A knapsack is defined as simply an integer representing its size.

```
class Knapsack // Defines the Knapsack Class
{
public:
   int capacity; // Identifies the capacity of the knapsack

   Knapsack(int knapsackSize) // Knapsack constructor
   {
      this->capacity = knapsackSize; // Set this knapsack's capacity to some given size
      std::cout << "[Constructed] Knapsack with capacity: " << std::setw(3) << this->capacity << std::endl;
   }
};
```

Listing 14: Knapsack Implementation

For all results, see the figures folder in this directory. In it you will find an images folder, containing the data parsed, and a results folder, containing the results from the operations performed on the data.