Assignment4

Sorin Macaluso

December 2023

1 Note

All code with comments is at the bottom in the appendix. The cout's in the original code had to be commented out or turned into a descriptive comment to explain what happened. All double quotes or single quotes where taken off and I added a comment with the data type next to it.

2 Main

2.1 Read File Knapsack

```
cout << \n;

vector<string> SpiceHeist;

string readInString;

ifstream File (spice.txt);

if (File.is_open()){

    while (File.good()){

        getline(File, readInString);

        SpiceHeist.push_back(readInString);


    }
    File.close();
}

else cout <<  Unable to open file ;

string SpiceName;
float SpicePrice;
int SpiceQty;
int count = 0;
int knapsackSize;

vector<int> kanpsacks;

for(string i: SpiceHeist){

    if(i.find( -- ) == std::string::npos){
        if(i.find( spice ) != std::string::npos){
            std::istringstream iss(i);
            std::string token;

```

```
                while (iss >> token) {
                    if (token ==  spice  || token ==  name  || token ==  =  || token ==  total_price  || token ==  qty ) {
                        continue;
                    }

                    if(token.find( ; ) != std::string::npos && count == 0){
                        SpiceName = token;
                        SpiceName.erase(remove(SpiceName.begin(), SpiceName.end(),  ; ), SpiceName.end());
                        count++;
                        continue;
                    }

                    if(token.find( ; ) != std::string::npos && count == 1){
                        token.erase(remove(token.begin(), token.end(),  ; ), token.end());
                        SpicePrice = stof(token);
                        count++;
                        continue;
                    }

                    if(token.find( ; ) != std::string::npos && count == 2){
                        token.erase(remove(token.begin(), token.end(),  ; ), token.end());
                        SpiceQty = stoi(token);
                        the spice after this
                        count = 0;
                        continue;
                    }

                }

                Spice(SpiceName, SpicePrice, SpiceQty);

            }else if(i.find( knapsack ) != std::string::npos){
                std::istringstream iss(i);
                std::string token;

                while (iss >> token) {
                    if (token ==  knapsack  || token ==  capacity  || token ==  = ) {
                        continue;
                    }

                    if(token.find( ; ) != std::string::npos){
                        token.erase(remove(token.begin(), token.end(),  ; ), token.end());
                        knapsackSize = stoi(token);
                        kanpsacks.push_back(knapsackSize);
                    }
                }

            }
        }
    }
```

There is a lot of different moving peaces involved in the piece of code that all comes together in order to make the correct logic needed to process the file for the spices and knapsacks. There are a lot of things that check to properly process the file for the spices. Using the spice word in the string to check if a new spice is added. Then since the values are in a order I made a simple counter to check which value we are on and adds the value to the proper value. This then passes the value to the Spice function which I will explain later in the lab. Then after all the spices are all done processing the knapsacks are all found, once the Knapp sack size is found that is passed to the fractional greedy algorithm and the algorithm runs for that Knapp sack size. The final thing to make note of in this code is that I have to trim the ; off since the stringstream only parses on space.

2.2 Read File Weighted Graph

```
    cout <<  \n ;

    vector<string> Graph;

    string readInString2;

    ifstream File2 ( graphs2.txt );

    if (File2.is_open()){

        while (File2.good()){

            getline(File2, readInString2);

            Graph.push_back(readInString2);


        }
         File2.close();
    }

    else cout <<  Unable to open file ;

    int Start = 0;
    int End = 0;
    int weight = 0;
    int VertexName = 0;
    int GraphCount = 0;
    bool BFGTest = false;
    int tell = 0;

    for(string i: Graph){

        if(i.find( -- ) == std::string::npos){
            if(i.find( new ) != std::string::npos || i == Graph.back()){

                if(tell == 0){
                    tell = 1;
                }else{

                    if(VertexName > 0){

                        BFGTest = BellmanFord();

                        if(BFGTest == false){
                            cout <<  \n ;
                            cout <<  There was a error in the shortest path
   calcualtion  <<  \n ;
                        }else{
                            cout <<  \n ;
                            cout <<  No error in calculating the shortest path  <<  \n
    ;
                        }

```

```
                        DeleteVertex();

                        Start = End = weight = VertexName = 0;
                    }else{
                        continue;
                    }
                }

            }else if(i.find( vertex ) != std::string::npos){

                std::istringstream iss(i);
                std::string token;

                while (iss >> token) {
                    if (token ==  add  || token ==  vertex ) {
                        continue;
                    }

                    VertexName = stoi(token);

                }

                Vertex(VertexName);

            }else if(i.find( edge ) != std::string::npos){

                std::istringstream iss(i);
                std::string token;

                while (iss >> token) {
                    if (token ==  add  || token ==  edge  || token ==  - ) {
                        continue;
                    }

                   if(GraphCount == 0){
                        Start = stoi(token);
                        GraphCount++;
                   }else if (GraphCount == 1){
                        End = stoi(token);
                        GraphCount++;
                   }else{
                        weight = stoi(token);
                        GraphCount = 0;
                   }

                }

                AddEdge(Start, End, weight);

            }
        }
    }
}
```

The processing for the graphs is a bit simpler than the spices. Since the order of the graph values are the new graph command, vertex's, and the edges. I first have to get all the vertex values and add them for the file and then handle the edges. For this kind of processing I again take advantage of the fact that the stringstream splits on spaces. This allows the three values for the edges (start, end, and weight) to be processed in the same way as the spices. There are not any ; to get out this time so I used the count method to tell which thing I am adding and then send that to the AddEdge method which will be explained later.

3 Fractional Greedy Algorithm

3.1 Spice and Knapsack

```
struct spice{
    string name;
    float price;
    int QTY;
    float unitPrice;
    bool Processed;
};

vector<spice*> SpiceHolder;

void Spice(string SpiceName, float SpicePrice, int SpiceQTY){

    spice* Spice = new spice;

    Spice->name = SpiceName;
    Spice->price = SpicePrice;
    Spice->QTY = SpiceQTY;
    Spice->unitPrice = SpicePrice/SpiceQTY;
    Spice->Processed = false;

    SpiceHolder.push_back(Spice);

}
```

This is the spice class that I use to hold the spices. The knapsack is a number that is passed to the algorithm. For the spices class I made the name, price, QTY, unit price (which is calculated with the price and the QTY), and then the Processed value so that I can tell if the spice has been processed or not (taken from Assignemnt3) for when the algorithm is running.

3.2 Fractional Greedy Algorithm

```
void FractionalGreedy(int size){
    int NapSize = size;
    const int trueSize =  size;
    float total = 0;

    vector<string> SpiceName;
    vector<int> SpiceAmount;

    while(NapSize > 0){
        spice* GUnitPrice = nullptr;

        for(spice* j : SpiceHolder){
            if(GUnitPrice == nullptr || j->unitPrice > GUnitPrice->unitPrice && j->Processed == false){
                GUnitPrice = j;
            }
        }

        if (GUnitPrice->Processed == true) {
            break;
        }

        GUnitPrice->Processed = true;

        if(GUnitPrice->QTY <= NapSize){
```

```

            NapSize = NapSize - GUnitPrice->QTY;
            total = total + GUnitPrice->price;

            SpiceName.push_back(GUnitPrice->name);
            SpiceAmount.push_back(GUnitPrice->QTY);


        }else if(GUnitPrice->QTY > NapSize || NapSize == 0){

            total = total + GUnitPrice->unitPrice * NapSize;

            SpiceName.push_back(GUnitPrice->name);
            SpiceAmount.push_back(GUnitPrice->QTY - NapSize);

            break;

        }

    }


    cout <<  Knapsack of capacity   << std::to_string(trueSize) <<   is worth   << setprecision(3) << total <<   quatloos and contains  ;

    for(int i = 0; i < SpiceName.size(); i++){
        cout << SpiceAmount[i] <<   scoops of   << SpiceName[i] <<  .  ;
    }

    cout <<  \n ;

    for(spice* i : SpiceHolder){
        i->Processed = false;
    }

}
```

This is a fractional greedy algorithm. What this will do is first tell the greatest unit price of all the spices that are not currently processed. Once the greatest unit price is determined then the next is to determine whether its a fraction or a whole add. This is done with the if else statement, the first if tells if its a whole add by saying if the spice quantity is less than or equal to the current Knapp Sack size (NapSize). In this if statement if the condition is meet then this will add the whole quantity to the total. Then the quantity that has been added and subtract it from the NapSize so that we have a new total NapSize that we need to check on. For the else if statement that checks the fraction this will be check by seeing if the quantity from the greatest unit price is greater than the current NapSize, or is 0 (meaning the NapSize is empty). If this condition is meet then the total has the NapSize times the unit price of the greatest unit price. Then all the data is passed back, this also happens for the if statement. The data that is pushed back is the name and the quantity, this is used to keep track of what has been added to the Knapp sack, for the else if the quantity added is the greatest unit price spice quantity minus the Napsize, so that we get the fraction needed to fill the rest of the NapSize. After the program will break as it is done processing the value. Finally the program will output the data so that it can be viewed in terminal, and also reset all the Processed values for the spices to be used for the next Knapp sack size.

The run time for this algorithm is O(n^2). Where n is the Knapp Sack Size this occurs because there is a while loop that at worst case needs to go through every spice as well as the for loop to check for the greatest unit price. This makes it a O(n^2) since at the worst case you have to go through all the spice's in order to fill the whole Knapp Sack. As well as go through all the spice in order to check which spice is used to be the greatest unit price. After that is all calculated the greatest unit price spice is added to the Knapp sack and then the loop continues. The run time can be improved if I was to sort the SpiceHolder vector by greatest unit price. If I did this the run time could be of time O(nlog(n)) + O(n) (merge or quick sort), or O(n^2) + O(n) (one for a linear or selection sort, then other for the Knapp sack size).

4 Weighted Graphs

4.1 New Graph Class as Linked Objects

```
struct Graph{
    int Vertex;
    vector<int> neighbors;
    vector<int> weights;
    int Distance = 8675309;
    vector<int> BackToTheFuture;
};
```

This is the class that I use to hold the directed weighted graph as a linked object. I have the vertex, a vector for the neighbors, the weights, the default distance for the Belllman Ford Single Source Shortest Path algorithm. Then the BackToTheFuture vector to hold the path that is taken for the single source shortest path.

4.2 Functions for Graph

```
vector<Graph*> VertexHolder;

void Vertex(int VertexName){
    Graph* vertex = new Graph;
    vertex->Vertex = VertexName;
    VertexHolder.push_back(vertex);
}

void AddEdge(int Start, int End, int Weight){
    VertexHolder[Start-1]->neighbors.push_back(End);
    VertexHolder[Start-1]->weights.push_back(Weight);

}

void DeleteVertex(){
    VertexHolder.clear();
}
```

These are all the functions that are needed in order to run the program correctly. The Vertex function will simply create the object of the class Graph, and add the vertex name that is passed to the function to the vertex value of the object of the class. Then add that newly created object of the Graph class to the VertexHolder vector. Then once the edges and weight for that read in set of edges is found the AddEdges function is called in order to add that data to the proper Vertex object that is stored in the VertexHolder vector. Finally the DeleteVertex is needed to clear the previous Vertex and edges added out of the old graph to set up for the new graph that will be made.

4.3 Bellman Ford Single Source Shortest Path

```
bool BellmanFord(){//graph, weight, source

    IniatSS();

    for(int s = 0; s < VertexHolder.size()-1 ; s++){
        for(int o = 0; o < VertexHolder.size(); o++){
            for(int r = 0; r < VertexHolder[o]->neighbors.size(); r++){
                int neighbor = VertexHolder[o]->neighbors[r];
                int weight = VertexHolder[o]->weights[r];
                Relax(VertexHolder[o]->Vertex, neighbor, weight);
            }
        }
    }

    for(int i = 0; i < VertexHolder.size()-1 ; i++){
```

```
        for(int n = 0; n < VertexHolder[i]->neighbors.size(); n++){
            int neighbor = VertexHolder[i]->neighbors[n];
            int weight = VertexHolder[i]->weights[n];
            if(VertexHolder[i]->Distance != 8675309 && (VertexHolder[i]->Distance + weight) < VertexHolder[neighbor-1]->Distance){
                return false;
            }
        }
    }

    cout <<  \n ;
    for(int t = 1; t < VertexHolder.size(); t++){
        cout <<  The path from   << VertexHolder[0]->Vertex <<   -->   << VertexHolder[t]->Vertex <<   is   << VertexHolder[0]->Vertex <<   -->  ;
        for(int o = 0; o < VertexHolder[t]->BackToTheFuture.size(); o++){
            if(o == VertexHolder[t]->BackToTheFuture.size()-1){
                cout << VertexHolder[t]->BackToTheFuture[o] <<  \n ;
            }else if(o < VertexHolder[t]->BackToTheFuture.size()){
                cout << VertexHolder[t]->BackToTheFuture[o] <<   -->  ;
            }
        }
    }

    return true;
```

For the Bellman Ford Single Source Shortest Path the very first step is to initialize the start. I have the IniatSS() function handle this for me. Then the algorithm start to work. This is started in the three nested for loops that are shown. These are really only needed because of how I processed the data for myself. the first loop will go through all the Vertex minus the last one. Then go through all the vertexes again, and the final for loop will be used to get the neighbors for that particular vertex. Then the neighbor and the weight are found and passed to the Relax function to be "relaxed" (changed if there distance is smaller than current distance). Then once that goes through all the Vertex and the edges then we move to the next nested for loop. This for loop is to check for negative weight cycles. If this is found then false is returned to main, and in main a error message will appear to show that there was a error in the build of the single source shortest path. The other for loops at the end of the algorithm are used to give a nice output onto terminal so that it is easy to read the single source shortest path.

For the time complexity it is O(V * E). Where V is the number of vertices that are in the set and E is the number of edges in the set. With all of these for loops the time complexity is O(V * E) because at the very worst, the algorithm will need to go through all the vertices and the edges in order to determine the single source shortest path from the source to the other vertices. This will be the only case as well because the algorithm needs to check all the edges and weights so that it can have the most accurate single source shortest path for each of the vertices. Since going through all vertices and edges happened 3 time in the algorithm the true time complexity is O(3(V * E)). But we remove constants so the time complexity is O(V * E).

4.4 Initialize

```
void IniatSS(){
    VertexHolder[0]->Distance = 0;

}
```

This function is used in the Bellman Ford Single Source Shortest Path in order to properly set up the algorithm. For the algorithm to work the start Vertex needs to have a distance of 0 in order for the relax function to be able to start comparing the values together.

4.5 Relax

```
void Relax(int start, int end, int weight){//comeing from, going too, weight
    if(VertexHolder[start-1]->Distance != 8675309 && (VertexHolder[start-1]->Distance + weight) < VertexHolder[end-1]->Distance){
        VertexHolder[end-1]->Distance =  VertexHolder[start-1]->Distance + weight;
```

```
        VertexHolder[end-1]->BackToTheFuture = VertexHolder[start-1]->BackToTheFuture;
        VertexHolder[end-1]->BackToTheFuture.push_back(VertexHolder[end-1]->Vertex);
    }
}
```

This is the most important function to the Bellman Ford Single Source Shortest Path algorithm. This function is so important because it it what changes the distance values to show what the single shortest path from the start vertex to the other vertex is. It changes the distance to store the shortest path, as well as updates the BackToTheFuture vector to have the path that should be fallowed so that I could show in terminal the shortest path.

This happens with the if statement, it checks if the start distance does not equal the max distance. It also checks if the start values distance + the weight is less than the end distance. Since the start has the current shortest distance + the new weight is less than the end values distance. If so then we showed add it to the distance so that we have a new updates shortest path. As well as add that data to BackToTheFuture vector so that we can keep track of where we are coming from. Before that we must insure that the end and start BackToTheFuture vectors are the same so that it is all unanimous.

5 References

Geeks For Geeks Min and Max of Vector
Geeks For Geeks Fractional Knapsack
I used the first link to help find the greatest unit price of something. Then the second linked to figure out the basics of what I would need to solve the fractional greedy sort problem.

Stack Overflow substring in string
tutorialspoint
This linked was used to look at how to get the substring of a string for token so that I could get rid of the semi colon at the end for the token before adding it for the spice data. The second link was also used to help solve the problem of getting rid of the semi colon before processing it as data

programiz string to float
This link was use to turn a string into a float and other data values. This was used so that the correct data type is assigned to the value not skipped by token so before it is processed as the correct data type when it is added to the value in the class.

6 Appendix

6.1 Main.cpp

```
//first two are librarys
#include <iostream> //object oriented library that allows input and output using streams
#include <fstream> //allows for the reading of a file in the library
#include <string> /* These three are used for the removing of a space for the strings */
#include <algorithm> /* These three are used for the removing of a space for the strings */
#include <cctype> /* These three are used for the removing of a space for the strings */
#include <iomanip> //used to set the amount of accuracy for the decmial points
#include <vector>
#include <sstream>
#include FractionalGreedy.hpp
#include DirectedWeightedGraph.hpp

using namespace std;

//biggest issue was getting the relax function to do the proper comparison
//how i have it set up causes a lot of confusion on what to add the weight to and what to compare the distance to
```

```


//main functions
int main(){

    cout << \n;

    vector<string> SpiceHeist;

    //start of the file stream studd
    string readInString;

    //opens the right file
    ifstream File (spice.txt);

    //checks if the file is open
    if (File.is_open()){

        //while file is open gets the line
        while (File.good()){

            getline(File, readInString);

            //add the now properly formatted line to the array
            SpiceHeist.push_back(readInString);


        }
        //closed the file at the end when all done
        File.close();
    }

    //error checking if the file is not opened
    else cout <<  Unable to open file ;

    //variables that will be used for processing
    string SpiceName;
    float SpicePrice;
    int SpiceQty;
    int count = 0;
    int knapsackSize;

    //will hold the knapsacks value
    vector<int> kanpsacks;

    //start of the long amount of logic that is used to correctly process the file
    for(string i: SpiceHeist){

        //checks for comments
        if(i.find( -- ) == std::string::npos){
            //the word spice it found then we know that there is a new spice
            if(i.find( spice ) != std::string::npos){
                std::istringstream iss(i);
                std::string token;

                while (iss >> token) {
                    //then we skip over all the words and symbols so that we are left with just the values
                    if (token ==  spice  || token ==  name  || token ==  =  || token ==  total_price  || token ==  qty ) {
```

```
                    continue;
                }
                //this only works is the values stay in the same format

                //the first thing is always the spice name so we find the token with the semi colon
                if(token.find( ; ) != std::string::npos && count == 0){
                    //make that token the SpiceName
                    SpiceName = token;
                    //trim the semi colon off
                    SpiceName.erase(remove(SpiceName.begin(), SpiceName.end(),  ; ), SpiceName.end());
                    //make the count one more so that we can move on to the next thing to add
                    count++;
                    continue;
                }

                //the second thing is always the SpicePrice
                if(token.find( ; ) != std::string::npos && count == 1){
                    //since this will be a float we have to get ride of the semi colon first before converting from string to float
                    token.erase(remove(token.begin(), token.end(),  ; ), token.end());
                    //convert
                    SpicePrice = stof(token);
                    //move the count along to check for the next thing
                    count++;
                    continue;
                }

                //the third this will always be the SpiceQTY
                if(token.find( ; ) != std::string::npos && count == 2){
                    //since this wil be a int we have the trim the semi colon off first before converstion
                    token.erase(remove(token.begin(), token.end(),  ; ), token.end());
                    //convertion
                    SpiceQty = stoi(token);
                    //set the count back to look for the SpiceName since we have all the data for the spice after this
                    count = 0;
                    continue;
                }

            }

            //then pass that data to Spice function in  FractionalGreedy.hpp  to create the spice
            Spice(SpiceName, SpicePrice, SpiceQty);

        //if we see a knapsack word do other actions
        }else if(i.find( knapsack ) != std::string::npos){
            std::istringstream iss(i);
            std::string token;

            //much less to skip over
            while (iss >> token) {
                if (token ==  knapsack  || token ==  capacity  || token ==  = ) {
                    continue;
```

```
                }

                //same logic as before once the token have a semicolon at it thats the data
                if(token.find( ; ) != std::string::npos){
                    //trim the semi colon off since the knap sack size will be a int
                    token.erase(remove(token.begin(), token.end(),  ; ), token.end());
                    //convert
                    knapsackSize = stoi(token);
                    //add to the vector that will hold all the knap sack sizes
                    kanpsacks.push_back(knapsackSize);
                }
            }

        }
    }
}

//this wil go through all the knap sack sizes and send them over to the FractionGreedy algorithm in  FractionalGreedy.hpp
for(int i : kanpsacks){
    FractionalGreedy(i);
}

cout <<  \n ;

vector<string> Graph;

//start of the file stream studd
string readInString2;

//opens the right file
ifstream File2 ( graphs2.txt );

//checks if the file is open
if (File2.is_open()){

    //while file is open gets the line
    while (File2.good()){

        getline(File2, readInString2);

        //add the now properly formatted line to the array
        Graph.push_back(readInString2);


    }
    //closed the file at the end when all done
    File2.close();
}

//error checking if the file is not opened
else cout <<  Unable to open file ;

//values to to hold data and other output
int Start = 0;
int End = 0;
int weight = 0;
```

```
    int VertexName = 0;
    int GraphCount = 0;
    bool BFGTest = false;
    int tell = 0;

    //goes through the whole graph vector
    for(string i: Graph){

        //checks for comments
        if(i.find( -- ) == std::string::npos){
            //checks for new graph
            if(i.find( new ) != std::string::npos || i == Graph.back()){
                //will then go in to doing the work (calling the Bellman ford algo with data stored in the file
                if(tell == 0){ //not on first instance of new (cause nothing is there)
                    tell = 1;
                }else{
                    //and only if there is stuff in the vertex
                    if(VertexName > 0){

                        BFGTest = BellmanFord();

                        if(BFGTest == false){
                            cout <<  \n ;
                            cout <<  There was a error in the shortest path calcualtion  <<  \n ;
                        }else{ //BFGTest == true
                            cout <<  \n ;
                            cout <<  No error in calculating the shortest path  <<  \n ;
                        }

                        //will also reset everything
                        DeleteVertex();

                        Start = End = weight = VertexName = 0;
                    }else{
                        continue;
                    }
                }

            //then checks for vertex to see if we are adding a new vertex
            }else if(i.find( vertex ) != std::string::npos){

                std::istringstream iss(i);
                std::string token;

                //splits to find the vertex (seperation via space)
                while (iss >> token) {
                    if (token ==  add  || token ==  vertex ) {
                        continue;
                    }

                    VertexName = stoi(token);

                }

                Vertex(VertexName);

            //see if a edge is being added
```

```
            }else if(i.find( edge ) != std::string::npos){

                std::istringstream iss(i);
                std::string token;

                //splits on spaces so the edge will look like this 2 3 4
                while (iss >> token) {
                    if (token ==  add  || token ==  edge  || token ==  - ) {
                        continue;
                    }

                    //count is used to tell what is the first and second vertex and the weight of that path
                   if(GraphCount == 0){ //first
                        Start = stoi(token);
                        GraphCount++;
                   }else if (GraphCount == 1){ //second
                        End = stoi(token);
                        GraphCount++;
                   }else{//weight
                        weight = stoi(token);
                        GraphCount = 0;
                   }

                }

                //then adds it
                AddEdge(Start, End, weight);

            }
        }
    }



}
```

6.2 FractionalGreedy.hpp

```
//first two are librarys
#include <iostream> //object oriented library that allows input and output using streams
#include <fstream> //allows for the reading of a file in the library
#include <string> /* These three are used for the removing of a space for the strings */
#include <algorithm> /* These three are used for the removing of a space for the strings */
#include <cctype> /* These three are used for the removing of a space for the strings */
#include <iomanip> //used to set the amount of accuracy for the decmial points
#include <vector>
#include <sstream>

using namespace std;

//this is the spice class that i made
struct spice{
    //name of the spice
    string name;
```

```
    //the price as a float if needed
    float price;
    //the QTy
    int QTY;
    //unitPrice to be caculated
    float unitPrice;
    //processed (taken from Assignment3) so that we do not indefinetly check the same
    spice
    bool Processed;
};

//this will hold all the different spice s
vector<spice*> SpiceHolder;

void Spice(string SpiceName, float SpicePrice, int SpiceQTY){

    //creates a new spice
    spice* Spice = new spice;

    //uses data to populate the parts of the class
    Spice->name = SpiceName;
    Spice->price = SpicePrice;
    Spice->QTY = SpiceQTY;
    Spice->unitPrice = SpicePrice/SpiceQTY;
    Spice->Processed = false;

    //then adds the pointer to the spice to the SpiceHolder vector so that i can
    access to them all
    SpiceHolder.push_back(Spice);

}


void FractionalGreedy(int size){
    //is given the size of the knap sack
    int NapSize = size;
    //consant for the print out
    const int trueSize =  size;
    //to keep hold of the total
    float total = 0;

    vector<string> SpiceName;
    vector<int> SpiceAmount;

    //will go until the Knap Sack is 0 in size
    while(NapSize > 0){
        //create a new pointer to find the greatest unit price
        spice* GUnitPrice = nullptr;

        //goes through all the spices that are in the spice holder
        for(spice* j : SpiceHolder){
            //if there is a new greatest unit price
            if(GUnitPrice == nullptr || j->unitPrice > GUnitPrice->unitPrice && j->
    Processed == false){
                //make the GUnitPrice pointer equal to it
                GUnitPrice = j;
            }
        }

        if (GUnitPrice->Processed == true) {
```

```
            //no more spices left
            break;
        }

        //will set the spice of the greatest one to true so that if it is used up in full we do not check it
        //of if its a fraction then we know we are done
        GUnitPrice->Processed = true;

        if(GUnitPrice->QTY <= NapSize){

            //will handle whole amounts

            NapSize = NapSize - GUnitPrice->QTY;
            total = total + GUnitPrice->price;

            //will add the amounts to a vector so that we have access to all the spices that made up the greatest combination (whole amounts)
            SpiceName.push_back(GUnitPrice->name);
            SpiceAmount.push_back(GUnitPrice->QTY);


        }else if(GUnitPrice->QTY > NapSize || NapSize == 0){

            //will handle fractional amounts

            total = total + GUnitPrice->unitPrice * NapSize;

            //will add the amounts to a vector so that we have access to all the spices that made up the greatest combination (fractional amounts)
            SpiceName.push_back(GUnitPrice->name);
            SpiceAmount.push_back(GUnitPrice->QTY - NapSize);

            break;

        }

    }


    //will handle the out put onto terminal
    cout <<  Knapsack of capacity   << std::to_string(trueSize) <<   is worth   << setprecision(3) << total <<   quatloos and contains  ;

    //goes through the vecotrs that hold the data and prints out everything in it
    for(int i = 0; i < SpiceName.size(); i++){
        cout << SpiceAmount[i] <<   scoops of   << SpiceName[i] <<  .  ;
    }

    //creates a new line for the next input
    cout <<  \n ;

    //resets all the spice values Prcosessed values to false so that it can be done again with a different knap sack
    for(spice* i : SpiceHolder){
        i->Processed = false;
    }

}
```

6.3 DirectedWeightedGraph.hpp

```
//first two are librarys
#include <iostream> //object oriented library that allows input and output using streams
#include <fstream> //allows for the reading of a file in the library
#include <string> /* These three are used for the removing of a space for the strings */
#include <algorithm> /* These three are used for the removing of a space for the strings */
#include <cctype> /* These three are used for the removing of a space for the strings */
#include <iomanip> //used to set the amount of accuracy for the decmial points
#include <vector>
#include <sstream>

//hardest thing was wrapping my head around what the relax function is compariring to make the graph RELAX

using namespace std;

//class to hold the vertex
struct Graph{
    //name of the vertex
    int Vertex;
    //will hold the neighbors for that vertex
    vector<int> neighbors;
    //will hold the weights that go along with that neighbors
    vector<int> weights;
    //this is the default distance made and then will change with relax
    int Distance = 8675309;
    //this will hold the shortest path from the source to desitnation
    vector<int> BackToTheFuture;
};

//this will hold the vertex pointers
vector<Graph*> VertexHolder;

//since the graph2.txt file reads in the vertex s first i build the vertex with the class and store the id
//then put it into the VertexHolder
void Vertex(int VertexName){
    //creats the pointer to the Graph object called vertex
    Graph* vertex = new Graph;
    //adds the VertexName as the Vertex for the pointer
    vertex->Vertex = VertexName;
    //adds to the VertexHolder vector
    VertexHolder.push_back(vertex);
}

//this will add the end vertex that is found for the edges as a neighbor of the start vertex
//as well as the corresponding weight (since the edge has add edge start - end weight)
void AddEdge(int Start, int End, int Weight){
    //adds end as a neighbor to the start
    VertexHolder[Start-1]->neighbors.push_back(End);
    //as well as the weight corresponding it
    VertexHolder[Start-1]->weights.push_back(Weight);

}
```

```cpp

void DeleteVertex(){
    //will clear the vertex in VertexHolder to have a new graph
    VertexHolder.clear();
}

void IniatSS(){//graph, source
    //initalizes the source vertex
    //always the first vertex
    VertexHolder[0]->Distance = 0;

}

void Relax(int start, int end, int weight){//comeing from, going too, weight
    //will check if the start-1 (have to get array index for the vertex) distance is not the greatest number
    //And that the start->distance plus the weight is less than the end->distance
    if(VertexHolder[start-1]->Distance != 8675309 && (VertexHolder[start-1]->Distance + weight) < VertexHolder[end-1]->Distance){
        //if so the end distance is the start distance plus weight
        VertexHolder[end-1]->Distance =  VertexHolder[start-1]->Distance + weight;
        //will equat the end to the start BackToTheFuture vector so that they we maintain the exisitng path
        VertexHolder[end-1]->BackToTheFuture = VertexHolder[start-1]->BackToTheFuture;
        //then adds the end value to the end BackToTheFutre vector
        VertexHolder[end-1]->BackToTheFuture.push_back(VertexHolder[end-1]->Vertex);
    }

}

bool BellmanFord(){//graph, weight, source

    //calls the inilize function to make sure everything is set up
    IniatSS();

    //goes through all vertex except one since the frist is the source one
    for(int s = 0; s < VertexHolder.size()-1 ; s++){
        //then goes through all vertex
        for(int o = 0; o < VertexHolder.size(); o++){
            //and all there neighbors (to go through all added edges)
            for(int r = 0; r < VertexHolder[o]->neighbors.size(); r++){
                //gets the neighbor
                int neighbor = VertexHolder[o]->neighbors[r];
                //gets the weight
                int weight = VertexHolder[o]->weights[r];
                //then passes it to relax to see if the relax should happen or not
                Relax(VertexHolder[o]->Vertex, neighbor, weight);
            }
        }
    }

    //goes through al vertex s
    for(int i = 0; i < VertexHolder.size()-1 ; i++){
        //then goes though all the neighbors
        for(int n = 0; n < VertexHolder[i]->neighbors.size(); n++){
            //find the neighbors again
            int neighbor = VertexHolder[i]->neighbors[n];
            //finds the weight again (for all edges)
            int weight = VertexHolder[i]->weights[n];
            //checks for negative weight cycle for the algorithm
```

```
            if(VertexHolder[i]->Distance != 8675309 && (VertexHolder[i]->Distance + weight) < VertexHolder[neighbor-1]->Distance){
                //will return false if found
                return false;
            }
        }
    }

    // else will Print the path
    //formatting
    cout <<  \n ;
    //goes through all Vertex in VertexHolder
    for(int t = 1; t < VertexHolder.size(); t++){
        //prints out start of the path string
        cout <<  The path from   << VertexHolder[0]->Vertex <<   -->   << VertexHolder[t]->Vertex <<   is   << VertexHolder[0]->Vertex <<   -->  ;
        for(int o = 0; o < VertexHolder[t]->BackToTheFuture.size(); o++){
            //then will go through the BackToTheFuture Vextor to give the path way
            if(o == VertexHolder[t]->BackToTheFuture.size()-1){
                //for a clean output
                cout << VertexHolder[t]->BackToTheFuture[o] <<  \n ;
            }else if(o < VertexHolder[t]->BackToTheFuture.size()){
                //or this to get the arrow
                 cout << VertexHolder[t]->BackToTheFuture[o] <<   -->  ;
            }
        }
    }

    //return true so that in main can be checked
    return true;



}
```