Assignment Four

Collin Drake
Collin.Drake1@Marist.edu

December 9, 2023

1 Directed Graphs

1.1 The Data Structure

A graph is a data structure made up of a collection of objects, denoted as vertices, and the edges or links that connect them. These vertices, or points, in a graph, contain three components: a unique identifier (ID), a distance, and a pointer to its predecessor. There are two types of graphs, directed and undirected. Each kind defines how the edges interact between vertices. In a directed graph, each edge establishes a clear direction from one vertex to another. However, in an undirected graph, edges have no direction, showing symmetrical links between vertices. In our case, we will be looking at a directed graph. Directed graphs are commonly represented in an adjacency list manner showing the distance it takes to travel from one vertex to all other vertices within the graph. Three fundamental operations make up a directed graph: addVertex, addEdge, and findVertexByID.

addVertex

```
    void Graphs::addVertex(string id){ //add vertex to graph object
        //creates a new vertex and adds it to the graph
        Vertex* newVertex = new Vertex(id);
        graph.push_back(newVertex);
    }
```

addEdge

```
    void Graphs::addEdge(string from, string to, int cost){
    //adds an edge to the graph object
        //finds the vertex's linked to the given ids
        Vertex* fromVertex = findVertexByID(from);
        Vertex* toVertex = findVertexByID(to);
        //creates a new edge
        Edge* newEdge = new Edge(fromVertex, toVertex, cost);
        //adds it to graph edge vector and to the source vertex
        edges.push_back(newEdge);
    }
```

findVertexByID

```
    Vertex* Graphs::findVertexByID(string id){ //searches the graph object for a given vertex
        //returns the pointer to the vertex within the graph object
        for(int i = 0; i < graph.size(); i++){
            if(graph[i]->id == id){
                return graph[i];
            }
        }
        return nullptr;
    }
```

1.2 Asymptotic Analysis

Given a directed graph containing vertices and edges, two fundamental operations can be performed: adding a vertex and adding an edge. Both functions can be characterized by their time complexities; O(1) and O(n). In our case, to keep track of all vertices in the graph, we will use a vector that contains pointers to each vertex object. When adding a vertex to the graph, the worst-case time complexity would be constant time, or O(1), as the new vertex would be added to the end of the graph object. The operation of adding an edge between two vertices would be similar, except its time complexity would be linear time, or O(n), as the vertices that make up the edge would have to be discovered by traversing the graph object. Given there are two vertices per edge, adding an edge would run in O(2n) time because you would have to traverse the graph object twice searching for each vertex. However, if we throw away constants, the resulting time complexity would be linear time, or O(n). Once the edge is created it is added to the edge vector which keeps track of all the edge objects using pointers.

2 Bellman-Ford SSSP

2.1 The Algorithm

The Bellman-Ford Single-Source Shortest Path algorithm is commonly used to find the quickest path from one vertex to all the other vertices within a directed graph. Three functions allow this algorithm to work: Bellman-Ford, initSSSP, and Relax. The Bellman-Ford function is the source of this algorithm and it calls all the other functions. Within the Bellman-Ford algorithm, there are three main goals: call initSSSP to initialize each vertex within the graph, call relax to relax all of the edges within a graph and set each involved vertices predecessor, and check for negative weight cycles. A negative weight cycle is a problem that occurs in the SSSP algorithm when the algorithm keeps updating the distances to vertices within the cycle in a way that never leads to a solution.

Bellman-Ford

```
bool Graphs:: bellmanFord(){ //sssp algorithm
    //create source vertex pointer
    Vertex* source = graph[0];
    //call initialize function
    initSSSP(source);
    //iterate through all vertices in the graph
    for(int i = 0; i < graph.size() - 1; i++){
        //for each edge in the graph call relax
        for(int j = 0; j < edges.size(); j++){
            Edge* edge = edges[j];
            //call relax
            relax(edge);
        }
    }
    //check for negative cycles
    for(int k = 0; k < edges.size(); k++){
        Edge* edge = edges[k];
        if(edge->to->distance > edge->from->distance + edge->cost){
            return false;
        }
    }
    return true;
}
```

initSSSP

```
void Graphs:: initSSSP(Vertex* source){ //initialize everything
    for(int i = 0; i < graph.size(); i++){
        //for each vertex clear its predecessors and set its distance to large int
        Vertex* vertex = graph[i];
        vertex->distance = 8675309;
        vertex->predecessor = nullptr;
    }
    //source vertex has a distance of zero
    source->distance = 0;
}
```

Relax

```
void Graphs:: relax(Edge* edge){ //relax edges
    //add edges to predecessor vector
    if (edge->to->distance > edge->from->distance + edge->cost) {
        edge->to->distance = edge->from->distance + edge->cost;
        edge->to->predecessor = edge->from;
    }
}
```

2.2 Asymptotic Analysis

For the Bellman-Ford Single-Source Shortest Path algorithm, the time complexity can be solved by examining each of its functions: initSSSP, Relax, and Bellman-Ford. The initSSSP algorithm traverses the graph object, initializing each vertex's variables. The time complexity of initSSSP is linear time, or O(v), where v represents the number of vertices within the graph. While the Relax function involves minor calculations, the process of calling it within Bellman-Ford introduces a nested for loop. This loop iterates through each vertex and edge in the graph, resulting in a time complexity of $O(|v| * |e|)$, where $|v|$ is the number of vertices and $|e|$ is the number of edges. Lastly, the Bellman-Ford function features a single for loop that iterates through each edge of the graph, checking for negative weight cycles. This process has a linear time complexity O(e), where e represents the number of edges. Combining these complexities gives us $O(v + |v| * |e| + e)$. However, constants are disregarded, leading to a simplified final time complexity of $O(|v| * |e|)$.

3 Fractional Knapsack

3.1 The Algorithm

The Fractional Knapsack algorithm is an algorithm that aims to maximize profit based on a given knapsack and its size. When given a set of items with weights and prices the Fractional Knapsack algorithm can determine the fractional amount of each item that can fit into the knapsack for the best overall outcome. In our case, we used spices, because "She who controls the spice controls the universe."

Spice Constructor

```
Spices:: Spices(string name, double price, int qty){ //spice class constructor
    spiceName = name;
    totalPrice = price;
    spiceQty = qty;
    unitPrice = price / qty;
    processed = false;
}
```

Fractional Knapsack

```
void Knapsack:: fractionalKnapsack(vector<Spices*> allSpices){
//fractional knapsack algorithm
 //keep track of current weight and the price of the knapsack
     double currentWeight = 0;
     double priceTotal = 0;
     //sort by unit price high to low
     Sorts sort;
     sort.mergeSort(allSpices, 0, allSpices.size() - 1);
     //iterate through spice array
     for(int i = 0; i < allSpices.size(); i++){
         //check if the current spice can completely fit in the knapsack
         if(currentWeight + allSpices[i]->spiceQty <= knapCapacity){
             //add the entire spice to the knapsack
             currentWeight += allSpices[i]->spiceQty;
             priceTotal += allSpices[i]->totalPrice;
             addItem(allSpices[i]);
         }
         //else add a fraction of the spice
         else{
             double remaining = knapCapacity - currentWeight;
             double fraction = remaining / allSpices[i]->spiceQty;
             priceTotal += allSpices[i]->totalPrice * fraction;
             //create a new spice object with the fraction and add it to the knapsack
             Spices* fractionSpice = new Spices(allSpices[i]->spiceName,
             allSpices[i]->totalPrice * fraction, remaining);
             addItem(fractionSpice);
             currentWeight = knapCapacity;
         }
     }
     //check how many items and output contents of the knapsack
     cout << "Knapsack␣of␣capacity␣" << knapCapacity << "␣is␣worth␣"
     << priceTotal << "␣quatloos␣and␣contains␣";
     for(int i = 0; i < items.size(); i++){
         if(i == items.size() - 1){
             cout << "and␣" << items[i]->spiceQty << "␣scoop␣of␣"
             << items[i]->spiceName << "\n";
         }
         else{
             cout << items[i]->spiceQty << "␣scoop␣of␣" << items[i]->spiceName << ",␣";
         }
     }
 }
```

3.2 Asymptotic Analysis

For the Fractional Knapsack algorithm, determining the time complexity involves calculating its two fundamental processes: sorting the spices based on their unit price and subsequently placing them in the knapsack. In our case, we employ merge sort to arrange the items in descending order according to their unit price. The time complexity of this merge sort is log-linear time, denoted as O(s log s), where the s represents the number of spices. The next step involves placing the items into the knapsack, which is accomplished by using a for loop to iterate through each spice stored within the items vector to find the best possible outcome. Combining these steps we get a time complexity of O(s log s + s). However, we throw away constant factors resulting in a final time complexity of log-linear, or O(s log s).

4 Appendix

Graph.cpp

```
//This file creates the Graphs class
#include "Graphs.hpp"

Graphs::Graphs(){} //graph object constructor

void Graphs::addVertex(string id){ //add vertex to graph object
    //creates a new vertex and adds it to the graph
    Vertex* newVertex = new Vertex(id);
    graph.push_back(newVertex);
}

void Graphs::addEdge(string from, string to, int cost){
    //adds an edge to the graph object
    //finds the vertex's linked to the given ids
    Vertex* fromVertex = findVertexByID(from);
    Vertex* toVertex = findVertexByID(to);
    //creates a new edge
    Edge* newEdge = new Edge(fromVertex, toVertex, cost);
    //adds it to graph edge vector and to the source vertex
    edges.push_back(newEdge);
}

Vertex* Graphs::findVertexByID(string id){ //searches the graph object for a given vertex
    //returns the pointer to the vertex within the graph object
    for(int i = 0; i < graph.size(); i++){
        if(graph[i]->id == id){
            return graph[i];
        }
    }
    return nullptr;
}

bool Graphs:: bellmanFord(){ //sssp algorithm
    //create source vertex pointer
    Vertex* source = graph[0];
    //call initialize function
    initSSSP(source);
    //iterate through all vertices in the graph
    for(int i = 0; i < graph.size() - 1; i++){
        //for each edge in the graph call relax
        for(int j = 0; j < edges.size(); j++){
            Edge* edge = edges[j];
            //call relax
            relax(edge);
        }
    }
    //check for negative cycles
    for(int k = 0; k < edges.size(); k++){
        Edge* edge = edges[k];
        if(edge->to->distance > edge->from->distance + edge->cost){
            return false;
        }
    }
    return true;
}

void Graphs:: initSSSP(Vertex* source){ //initialize everything
    for(int i = 0; i < graph.size(); i++){
        //for each vertex clear its predecessors and set its distance to large int
        Vertex* vertex = graph[i];
        vertex->distance = 8675309;
```

```
            vertex->predecessor = nullptr;
        }
        //source vertex has a distance of zero
        source->distance = 0;
    }

    void Graphs:: relax(Edge* edge){ //relax edges
        //add edges to predecessor vector
        if (edge->to->distance > edge->from->distance + edge->cost) {
            edge->to->distance = edge->from->distance + edge->cost;
            edge->to->predecessor = edge->from;
        }
    }


    void Graphs:: outputSSSPResults(){ //output the results of the sssp algorithm
        //create stack
        Stack stack;
        for (int i = 1; i < graph.size(); i++) {
            //iterate through all vertices with current Vertex
            and counters to help with formatting output
            cout << graph[0]->id << " -> " << graph[i]->id << " cost is "
            << graph[i]->distance << "; ";
            Vertex* current = graph[i];
            int countOne = 0;
            int countTwo = 0;
            while(current != nullptr){
                stack.push(current->id);
                current = current->predecessor;
                countOne++;
            }
            //output based on position
            cout << "path: ";
            while(!stack.isEmpty()){
                string prev = stack.pop();
                if(countTwo == countOne - 1){
                    cout << prev;
                }
                else{
                    cout << prev << " -> ";
                    countTwo++;
                }
            }
            cout << "\n";
        }
        cout << "\n";
    }
```

Spices.cpp

```
    //this file creates the spices, knapsacks, and sorts class
    #include "Spices.hpp"

    //Spices class below------------

    Spices:: Spices(string name, double price, int qty){ //spice class constructor
        spiceName = name;
        totalPrice = price;
        spiceQty = qty;
        unitPrice = price / qty;
        processed = false;
    }

```

```
//Knapsack classes below----------

Knapsack:: Knapsack(double capacity){ //knapsack class constructor
    knapCapacity = capacity;
}

void Knapsack:: addItem(Spices* spice){ //add an item to the knapsack
    //create new spice to separate pointers and adds to knapsack
    Spices* itemSpice = new Spices(spice->spiceName, spice->totalPrice, spice->spiceQty);
    items.push_back(itemSpice);
}

void Knapsack::clearKnapsack() { //clear the items stored in the knapsack
    for (int i = 0; i < items.size(); i++) {
        Spices* spice = items[i];
        delete spice;
    }
    items.clear();
}


void Knapsack:: fractionalKnapsack(vector<Spices*> allSpices){
//fractional knapsack algorithm
    //keep track of current weight and the price of the knapsack
    double currentWeight = 0;
    double priceTotal = 0;
    //sort by unit price high to low
    Sorts sort;
    sort.mergeSort(allSpices, 0, allSpices.size() - 1);
    //iterate through spice array
    for(int i = 0; i < allSpices.size(); i++){
        //check if the current spice can completely fit in the knapsack
        if(currentWeight + allSpices[i]->spiceQty <= knapCapacity){
            //add the entire spice to the knapsack
            currentWeight += allSpices[i]->spiceQty;
            priceTotal += allSpices[i]->totalPrice;
            addItem(allSpices[i]);
        }
        //else add a fraction of the spice
        else{
            double remaining = knapCapacity - currentWeight;
            double fraction = remaining / allSpices[i]->spiceQty;
            priceTotal += allSpices[i]->totalPrice * fraction;
            //create a new spice object with the fraction and add it to the knapsack
            Spices* fractionSpice = new Spices(allSpices[i]->spiceName,
            allSpices[i]->totalPrice * fraction, remaining);
            addItem(fractionSpice);
            currentWeight = knapCapacity;
        }
    }
    //check how many items and output contents of the knapsack
    cout << "Knapsack of capacity " << knapCapacity << " is worth " << priceTotal
    << " quatloos and contains ";
    for(int i = 0; i < items.size(); i++){
        if(i == items.size() - 1){
            cout << "and " << items[i]->spiceQty << " scoop of " << items[i]->spiceName
            << "\n";
        }
        else{
            cout << items[i]->spiceQty << " scoop of " <<
            items[i]->spiceName << ", ";
        }
    }
}

```

```cpp
    //Sorts classes below-----------

    //merge sort for use with spice vector
    void Sorts:: mergeSort(vector<Spices*>& allSpices, int start, int end){
        if(start >= end){
            return;
        }
        //find middle point, sort left, sort right, and merge the sorted arrays
        int middle = (start + end) / 2;
        mergeSort(allSpices, start, middle);
        mergeSort(allSpices, middle + 1, end);
        merge(allSpices, start, middle, end);
    }

    //merge sorted arrays together
    void Sorts:: merge(vector<Spices*>& allSpices, int start, int middle, int end){
        //declare left and right pointers, subArray
        int left = start;
        int right = middle + 1;
        vector<Spices*> subVec(end - start + 1);

        //iterate through sub array
        for(int i = 0; i < end - start + 1; i++){
            if(right > end){
                //add element from left side
                subVec[i] = allSpices[left];
                left++;
            }
            else if (left > middle)
            {
                //add element from right side
                subVec[i] = allSpices[right];
                right++;
            }
            else if (allSpices[left]->unitPrice >= allSpices[right]->unitPrice)
            {
                //add element from left side
                subVec[i] = allSpices[left];
                left++;
            }
            else{
                //add element from right side
                subVec[i] = allSpices[right];
                right++;
            }
        }
        //move subvector elements to main vector
        for(int j = 0; j < end - start + 1; j++){
            allSpices[start + j] = subVec[j];
        }
    }
```

5 References

5.1 Links

Below are the resources I have used to create simple, readable, and beautiful code.

- The textbook helped me with basic algorithm and data structure definitions: Algorithms textbook
- Your website helped me form and articulate descriptions for each data structure and algorithm used: Labouseur.com
- For detecting the end of the file: mathbits.com
- I used this to help create my edge class: cseweb Lecture
- Helped create my fractional knapsack algorithm: geeksforgeeks
- Remove semicolons from string: stackoverflow
- This helped me form my Bellman-Ford SSSP definition: geeksforgeeks
- This helped me form my knapsack definition: Tutorialspoint.com
- This helped me explain negative cycles: LinkedIN.com