Assignment Four

Christian Sarmiento

christian.sarmiento1@marist.edu

December 7, 2024

1 Introduction

Assignment Four focuses on creating a program that first builds a graph to conduct Bellman-Ford's Single Source Shortest Path (SSSP) algorithm and then constructs a greedy algorithm to collect spices in a knapsack from a text file. This program was developed using C++.

2 Bellman-Ford SSSP

Since the graph structure is overall the same from the undirected version of the graph, I will highlight the key changes that makes it possible to conduct Bellman-Ford SSSP.

2.1 Graph.cpp

The file graphs2.txt contains a few different graphs to build. We use Graph.cpp to create these graph objects. Since we are dealing with directed, weighted graphs, we must provide support for this when adding edges:

```
1  void Graph::addEdge(std::string vertex1ID, std::string vertex2ID, int weightVal) {
2
3      // Variables
4      Vertex* vertex1;
5      Vertex* vertex2;
6
7      // Get both of the verticies
8      vertex1 = getVertex(vertex1ID);
9      vertex2 = getVertex(vertex2ID);
10
11     // Add an edge between them if both vertices exist - directed edge
12     if (vertex1 && vertex2) {
13
14         // Create a tuple with the neighbor vertex and weight
15         std::tuple<Vertex*, int> weightedEdge(vertex2, weightVal);
16
17         // Add edge
18         vertex1->addNeighbor(weightedEdge);
19
```

```
20          // Create a tuple for edges and weights
21          std::tuple<Vertex*, Vertex*, int> edgeWeightPairs(vertex1, vertex2, weightVal);
22          myEdges.push_back(edgeWeightPairs);
23
24      } // if
25
26      else
27          std::cout << "Could not add an edge between " << vertex1ID << " and " << vertex2ID
                << "." << std::endl;
28
29  } // addEdge()
```

The key change in this class method is the addition of tuples to save the edge weights between two vertices (Lines 21-22). We first create a tuple that contains the starting & ending vertices as well as the weight for the edge between them. We then save this tuple to a new instance variable called myEdges, which saves all of these vertex/weight combonations for the graph in one vector for access in Bellman-Ford SSSP later.

2.2 bellmanFordSSSP()

This function is what conducts our single source shortest path algorithm:

```
1   void bellmanFordSSSP(Graph& graph, Vertex* startVertex) {
2
3       // Variables
4       std::vector<Vertex*>& graphVertices = graph.myVertices;
5       std::vector<std::tuple<Vertex*, Vertex*, int>>& graphEdges = graph.myEdges;
6       bool success = true;
7
8       // Initialize starting distances and predecessor vertex
9       initSingleSource(graph, startVertex);
10
11      // Iterate through the vertices
12      for (int i=1; i <= (graphVertices.size() - 1); i++) {
13
14          // Iterate through the edges
15          for (int j=0; j < graphEdges.size(); j++) {
16
17              relax(std::get<0>(graphEdges[j]), std::get<1>(graphEdges[j]), std::get<2>(
                    graphEdges[j]));
18
19          } // for j
20
21      } // for i
22
23      // Checking for negative weight cycle
24      for (int k=0; k < graphEdges.size(); k++) {
25
26          if (std::get<1>(graphEdges[k])->minDistance > (std::get<0>(graphEdges[k])->
                minDistance + std::get<2>(graphEdges[k])))
27              success = false;
28
29      } // for k
30
31      // Print results
32      if (success)
33          outputShortestPath(startVertex, graphVertices);
34
35      else
36          std::cout << "Negative weight cycle." << std::endl;
37
38
39  } // bellmanFordSSSP()
```

First, we use our initSingleSource() function (Line 9) to initialize all the shortest distances from the start vertex to a big number and the distance to the start vertex to 0. After that, we iterate through all the vertices and edges (Lines 11 - 21) to see all of the possible combinations of paths between all the vertices. To update these distances, we use the relax() function at line 17 to check if the distance of the end vertex (or the vertex we are going to) is greater than the distance of the vertex we are at plus the weight to get the end vertex. If it is, the shortest path is memoized, or saved, to that vertex object and the predecessor object for the vertex is updated to that of the start vertex in order to keep track of the path to get there. This process is repeated for all vertices and edges, resulting in the shortest path. Because of this, the time complexity for bellmanFordSSSP() is $O(V + E)$, where V is the number of vertices, and E is the number of edges. After this main process of the function runs, we check for negative weight cycles at lines 23-29. This is done to ensure that the algorithm is valid and isn't just stuck in one place of the graph where it is tricking the algorithm in thinking that going in circles decreasing the weight is the real shortest path. After this, if the algorithm is valid, we use these two functions to output our results:

```
1  void getPath(Vertex* vertex, std::vector<std::string>& pathVector) {
2
3      // Base case - null pointer for predecessor
4      if (vertex->predecessor != nullptr)
5          getPath(vertex->predecessor, pathVector);
6
7      // Add the vertex ID to the path vector - in order with recursion
8      pathVector.push_back(vertex->getID());
9
10 } // getPath()
11
12 void outputShortestPath(Vertex* initVertex, std::vector<Vertex*> vertices) {
13
14     // Variables
15     std::string outputString = "";
16     std::vector<std::string> idVector;
17
18     // Iterate through the vertices, not including the initial vertex
19     for (int i=1; i < vertices.size(); i++) {
20
21         // Build first part  of the string
22         outputString += initVertex->getID();
23         outputString += " --> ";
24         outputString += vertices[i]->getID();
25         outputString += " cost is ";
26         outputString += std::to_string(vertices[i]->minDistance);
27         outputString += "; path: ";
28
29         // Call recursive function to get the path
30         getPath(vertices[i], idVector);
31
32         // With populated vector, add ids to the output string
33         for (int j=0; j < idVector.size(); j++) {
34
35             if (j != idVector.size() - 1)
36                 outputString += idVector[j] + " --> ";
37
38             else
39                 outputString += idVector[j];
40
41         } // for j
42
43         // Clear the vector for next iteration
44         idVector.clear();
45
46         // Add newline character to not crowd the output
47         outputString += "\n";
48
```

```
49      } // for i
50
51      // Print final string
52      std::cout << outputString << std::endl;
53
54  } // outputShortestPath
```

For our bellmanFordSSSP() output, we use the outputShortestPath() function which uses the starting vertex and the list of vertices in the graph to traverse the shortest path to each vertex. An output string is dynamically built throughout the function, adding to it as we iterate through each vertex (Lines 22-27). To accurately show the information for the shortest path, we must first traverse that shortest path for each vertex. This is done with the getPath() function which recursively goes through the vertices by accessing each of the vertices' predecessors until we reach the start vertex. By manipulating the nature of recursion, we save the shortest path from start to finish by adding the ID to a vector. Since it is recursion, the addition of IDs only starts once we reach the start vertex, preserving the order. Back in our outputShortestPath() function, we then use this constructed vector to easily output the IDs in order from start to end.

3 Spice Heist

This section of the code deals with parsing out the information from a file and then constructing spice objects to then conduct a greedy algorithm. Here is the architecture of the Spice class:

```
1   class Spice {
2
3       public:
4
5           // Instance Variables
6           std::string mySpiceName = " ";
7           double myTotalPrice = 0.0;
8           int myQuantity = 0;
9           double myUnitPrice = 0.0;
10
11
12          // Constructors
13          Spice();
14          Spice(std::string newName);
15
16          // Getters
17          std::string getSpiceName() const;
18          double getTotalPrice() const;
19          int getQuantity() const;
20          double getUnitPrice() const;
21
22          // Setters
23          void setSpiceName(std::string newName);
24          void setTotalPrice(double newTotal);
25          void setQuantity(int newQuantity);
26
27          // Class Methods
28          void computeUnitPrice();
29
30  }; // Class Graph
```

This is a very basic class definition. It has instance variables to save the name, price, quantity, and unit price of the given spice. It also includes the standard functions such as getters and setters for these variables.There isn't a setter for the unit price since the computeUnitPrice will compute and save this information to our unit price instance variable for us. Our parsing uses these functions to fully build our class. After we get our neccessary information from our parsing, which creates a list of spice objects, we compute the unit price for

each of our spices. Once we have this, we then sort the list of spice objects by our unit price using QuickSort. Once this is done, we are ready for our greedy algorithm:

```
void greedySpiceCollection(std::vector<int> bags, std::vector<Spice> spiceObjects) {

    // Variables
    int currKnapsackSize = 0;
    int currSpiceQty = 0;
    double currUnitPrice = 0.0;
    double totalKnapsackCost = 0.0;
    std::string finalOutput = "";
    std::string scoopOutput = "";
    int validSpices = 0;
    int numCurrSpices = 0;

    // Iterate through the knapsack sizes
    for (int l=0; l < bags.size(); l++) {

        // Loop variables
        currKnapsackSize = bags[l];
        totalKnapsackCost = 0.0;
        finalOutput = "";
        scoopOutput = "";
        std::map<std::string, int> spiceScoops;
        std::ostringstream roundedKnapsackCost;

        // Start building string
        finalOutput += "Knapsack of capacity " + std::to_string(currKnapsackSize) + " is
            worth ";

        // Make sure there is still space in the knapsack
        while (currKnapsackSize > 0) {

            // Iterate through the sorted spices
            for (int m=0; m < spiceObjects.size(); m++) {

                // Initalize our constrains/counts
                currSpiceQty = spiceObjects[m].getQuantity();
                currUnitPrice = spiceObjects[m].getUnitPrice();
                spiceScoops[spiceObjects[m].getSpiceName()] = 0;

                // Make sure there is still spice to grab
                while (currSpiceQty > 0 && currKnapsackSize > 0) {

                    // Add a scoop
                    totalKnapsackCost += currUnitPrice;
                    currSpiceQty--;
                    currKnapsackSize--;
                    spiceScoops[spiceObjects[m].getSpiceName()]++;

                } // while

            } // for m

        } // while

        // Count the number of valid elements for output purposes
        // (a valid element is an element that has more than 0 scoops)
        for (const auto& n : spiceScoops)
            if (n.second > 0)
                validSpices++;

        // Iterate through our scoops to build our output
        for (auto n = spiceScoops.begin(); n != spiceScoops.end(); ++n) {

```

```
62              // Ensure it is worthwhile to display
63              if (n->second != 0) {
64
65                  numCurrSpices++;
66                  scoopOutput +=  std::to_string(n->second) + " scoops of " + n->first;
67
68                  // Add comma delimiter if it isn't the last element
69                  if (numCurrSpices < validSpices)
70                      scoopOutput += ", ";
71
72              } // if
73
74          } // for n
75
76          // Add total number of quatloos obtained for the knapsack
77          roundedKnapsackCost << std::fixed << std::setprecision(1) << totalKnapsackCost;
78          finalOutput += roundedKnapsackCost.str() + " quatloos and contains ";
79
80          // Add total number of scoops
81          finalOutput += scoopOutput + ".";
82
83          // Final output
84          std::cout << finalOutput << std::endl;
85
86      } // for l
87
88  } // greedySpiceCollection()
```

This function takes in a vector for our knapsack capacities as well as a the vector for our spice objects. We first iterate through our knapsack capacities (Line 14) since we want a line of output for each knapsack capacity. We then define a few variables inside this loop that will do things such as build our output and calculate certain terms such as the knapsack cost (Lines 16-22). We then start dynamically building our string at line 25. Using a while loop, we ensure that we are not over extending over our knapsack capacity while we are adding spice to our knapsack (Line 28). In doing so, we make sure that all of the work to add spice to the knapsack is only done if there is space to do so. The nature of greedy algorithms involves us adding. the most valuable stuff first, in this case the stuff being the highest unit price of the spices. We do this by iterating through our sorted spice objects vector at line 31. For output purposes, we use a hash map (courtesy of C++) to save our scoops of spice. This ensures we are capturing the proper information depending on the size of the knapsack (Line 36). From here, we use another while loop to add a scoop to our knapsack, only doing so if there is spice to scoop up and if there is still space in our knapsack (Lines 39-47). After iterating through all of this and capturing our information, we get ready to build our output (Lines 53-84). The first for loop (Lines 55-57) just counts the valid entries, or entries that have more than 0 scoops, in our hash map so that we can properly format our output with appropriate commas. Then, we iterate through our hash map again to build our output (Lines 60-74). This will only add to our output string if there are valid entries. After the loop, we find our total rounded knapsack cost and add that to our output string (Lines 76-81). At this point, we can output the string (Line 84). This is repeated for each knapsack. As per our textbook, the sorting part of our algorithm happens in $0(nlog_2 n)$ time. Since our algorithm is running for each element, that is another n times. We have to check if there is still space, so that check runs in $O(f(n))$ time. Therefore, our greedy algorithm has a time complexity of $O(nlog_2 n + nf(n))$ (CLRS Pg. 461).

4 References

- GeeksForGeeks - Tuples in C++
- Introduction to Algorithms, 3rd Edition
- Labouseur.com - Algorithms Slides
- ChatGPT - Personal Chat for C++ Syntax & Help