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 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.

```
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
{
   std::vector<std::vector<std::string>> graphs; // Declare graphs as a vector of vectors of strings
   std::ifstream file("./input/graphs2.txt");    // Declare file and open file at location

   if (file.is_open()) // If the file is open
   {
      std::vector<std::string> currentGraph; // Declare currentGraph as a vector of strings
      std::string line;                      // Declare line as a string
      while (std::getline(file, line))       // For every line in the file
      {
         if (line.empty() || line.front() == '-') // If the line is empty or a comment line
         {
            continue; // Skip this line
         }
         if (line == "new graph") // If the line declares a new graph
         {
            if (!currentGraph.empty()) // If the current graph storage is not empty
            {
               graphs.push_back(currentGraph); // Push the current graph back to the graphs vector
               currentGraph.clear();           // Clear the current graph vector for the next graph to be read
            }
            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.

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).

```
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.

```
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.

```
bool mapPathways(std::string &startID) // Constructs the pathways between vertices from a starting vertex
{
   Vertex *startVertex = search(startID); // Search for the starting vertex using it's ID
   if (!startVertex)                      // If the starting vertex is not found
   {
      std::cout << "Vertex #" << startID << " not found" << std::endl;
      return false;
   }
   // If the starting vertex is found
   startVertex->distance = 0;                     // Set the starting vertex's distance to 0
   for (int i = 0; i < vertices.size() - 1; i++) // For every item in the list of vertices - 1
   {
      for (Edge *edge : edges) // For every edge in edges
      {
         Vertex *startEdgeVertex = search(edge->startID); // Set the starting edge vertex to that edge's start vertex
         Vertex *endEdgeVertex = search(edge->endID);     // Set the ending edge vertex to that edge's end vertex

         if (endEdgeVertex->distance > startEdgeVertex->distance + edge->weight) // Relax function to check if a shorter path exists
         {
            endEdgeVertex->distance = startEdgeVertex->distance + edge->weight; // Setting the shortest path
            endEdgeVertex->predecessor = startEdgeVertex;                       // Setting the predecessor vertex
         }
      }
   }
   // After all vertices have been mapped out
   for (Edge *edge : edges) // For every edge in edges
   {
      Vertex *startEdgeVertex = search(edge->startID);                     // Set the starting edge vertex to that edge's start vertex
      Vertex *endEdgeVertex = search(edge->endID);                         // Set the ending edge vertex to that edge's end vertex
      if (endEdgeVertex->distance > startEdgeVertex->distance + edge->weight) // If a shorter path is found after, then there is a negative weight cycle
      {
         std::cout << "<< Graph contains a negative-weight cycle >>" << std::endl;
         return false;
      }
   }
   // If no shorter paths exist
```

```
        for (Vertex *vertex : vertices)
        {
            if (vertex->id != startID) // For all vertices not the start
            {
                std::cout << "#" << startID << std::setw(3) << " -> #" << vertex->id << " | Cost: " << std::setw(3) <<
       vertex->distance << " | Path: ";
                printPath(vertex->id); // Prints the path from the source to this vertex
                std::cout << std::endl;
            }
        }
        return true;
    }
};
```

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:

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

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.

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.

```
std::vector<std::string> parseInstructions() // Parse the data from spice.txt into a vector of strings
{
    std::vector<std::string> instructions;   // Declare vector of strings to hold instructions
    std::ifstream file("./input/spice.txt"); // Read and open the file

    if (file.is_open()) // If the file is open
    {
        std::string line;                 // Placeholder for current line
        while (std::getline(file, line)) // While there is a line in the file
        {
            if (line.empty() || line.front() == '-') // If the line is empty or a comment
            {
                continue; // Skip
            }
            // For remaining lines:
            instructions.push_back(line);
        }
    }
    else // If the file is not open
    {
        std::cout << "Error opening file" << std::endl;
    }
    return instructions;
}
```

Listing 5: Parsing File Implementation

TODO:

```
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

TODO:

```
// Algorithms ~ A.Labouseur, Assignment 4 - Connor Fleischman
#ifndef H_BUILD
#define H_BUILD

#include "./graph/UseGraph.h"
```

Listing 7: Building Implementation

TODO:

```
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

TODO:

4 Clean Up

```
Destroying graph #1
-- Edges Deleted --
-- Vertices Deleted --
```

TODO:

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

TODO:

5 Miscellaneous Implementations

TODO: