CMPT 435 - Fall 2020 - Dr. Labouseur

Assignment Five – Bellman-Ford, Spice, and Everything Nice
Bellman-Ford Single Source Shortest Path Algorithm, and a Greedy Solution to the Fractional Knapsack Problem

---

Sam Alcosser
Samuel.Alcosser1@Marist.edu

December 3, 2020

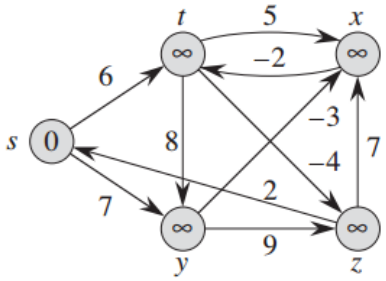
Contents

1 Introduction

For this documentation, there is a different format for how the processes are displayed. in this new format, the layout of each process will be outlined in an explanation, then all of the necessary data structures will be explained, and finally a step by step breakdown of the process itself will be given with direct references to line numbers.

To improve the flow of the document itself, smaller pieces such as header files, data structures, and supplementary methods will have outlines of their actual code in the appendix. This way, the reader can read at their own pace, depending on their level of prior knowledge of the topic.

2 Bellman-Ford Single Source Shortest Path (SSSP)



This algorithm works on weighted graphs. Weighted graphs are sets of vertices with one-way paths going between them, and each path, or edge, is assigned a weight. Although, not all vertices must be connected to each vertex, and edges can have negative weights. Also, all of the vertices hold a distance value which will be discussed later (For more explanation, see the Appendix for code explanations of vertices and edges) .

The Bellman-Ford SSSP algorithm, and single source shortest path algorithms in general, exactly what they sound like. They are ways to find the least costly and ”shortest path” from a ”single source” vertex to any of the other vertices in the graph. Bellman-Ford is a SSSP algorithm that is also useful as it can indicate whether or not there is a negative weight cycle. The process works in three main steps.

1. Start with all but the first vertex having a distance of ∞, as the first ”source” vertex will be set to a distance of 0.

2. ”Relax”, or update, the distances of the vertices using the weights of each edge. Repeat the updating process a number of times equivalent to the number of vertices in the graph to ensure the distances are correct.

3. Check for negative weight cycles by iterating over each edge one more time. If a negative weight cycle is found, return as false, otherwise, continue in order to return true.

Thankfully, the implementation of this code more or less is broken up by those three steps. To begin , there is the `init()` function.

Note: For a detailed explanation of how the data actually gets into the graph at the beginning, see the Appendix for the description of method `fileReader()`

2.1 `GraphManager::init()`: Initializing the Vertices

This first part of the algorithm is simply used to set up the initial distances of the vertices (hence the name `init()`).

```
1  void GraphManager::init(Vertex *source)
2  {
3          for (int i = 0; i < graph.size(); i++)
4          {
5                  graph[i]->distance = 10000000;
6                  graph[i]->prev = nullptr;
7          }
8          source->distance = 0;
9
10 }
```

The process in this function is very simple to see. It iterates a number of times equivilent to the size, or more specifically cardinality, of the graph's vertices(line 3). For each iteration it sets the indexed vertex to have a distance of 10000000, and a `nullptr` for its previous vertex (lines 5-6). The reason for the use of the number 10,000,000 and not infinity is that support of a real infinite value in C++ is very hit or miss.

When implementing one package which defines a global constant called `INIFINITY`, a quick printout shows that the values are so large that they seem to have rolled over into incredibly large negative numbers. So, after seeing that the real function of this value was to be a placeholder value that would undoubtedly be greater than any weight that could be defined, ten million was decided to be a safe value to imitate infinity. Finally, the source vertex is set to a distance of 0 (line 8).

2.2 `GraphManager::relax()`: Updating Distances Using the Edges

The second key aspect of this algorithm is the process of "relaxing" edges.

```
1  void GraphManager::relax(Edge *edge, Vertex *from, Vertex *dest)
2  {
3          if (dest->distance > from->distance + edge->weight)
4          {
5                  dest->distance = from->distance + edge->weight;
```

```
6                   dest->prev = from;
7           }
8   }
```

This process will take an edge and its two related vertices and attempt to update the destination vertex. The vertex will only be changed if the destination vertex's distance is greater than the sum of the distance of the `from` vertex and the `weight` of the edge combined (line 3). If this is the case, the distance of the destination vertex will be updated to that previously described sum, and it's previous vertex pointer will take the address of the from vertex (lines 5-6).

Realistically, it would've been possible to only pass the edge itself to this method. Although, within the process, it would have made for much dirtier code. There would need to be long chains of pointers to class members in many places. Therefore, all three parts are passed.

2.3 `GraphManager::Bellman()` : Bringing it All Together

Since the sub parts of this algorithm have been previously described, the algorithm itself should now make sense

```
1   bool GraphManager::Bellman()
2   {
3           init(graph[0]);
4           for (int i = 0; i < graph.size(); i++)
5           {
6                   for (Edge *e : weightMatrix)
7                   {
8                           relax(e, e->from, e->dest);
9                   }
10          }
11          for (int q = 0; q < weightMatrix.size(); q++)
12          {
13                  if (weightMatrix[q]->dest->distance > weightMatrix[q]->from->distance + weightMatrix[q]->weight)
14                  {
15                          cout << "Uh Oh! Found a negative cycle!" << endl;
16                          return false;
17                  }
18          }
19
20          for (int n = 1; n < graph.size(); n++)
21          {
22                  printPath(graph[0], graph[n]);
23          }
24
25          return true;
26  }
```

- Part 1 (line 3): initialize the graph with the `init()` method.
- Part 2 (lines 4-10): Relax using each edge using the `relax()` method a number of times equivalent to the cardinality of the set of vertices.

- Part 3 (lines 11-18): Test each edge for negative cycles. if the distance of the destination vertex is greater than the sum of the distance of the `from` vertex and the weight of the edge combined, report a negative cycle and return `false`.
- Part 4 (lines 20-23): For every non-source vertex, print the path from the source to that vertex using the `printPath()` method (described in detail in the appendix).
- Part 5 (line 25): Since the function would have been returned by this point if there was a negative cycle, we can assume that there are shortest paths to every point, so we return true.

2.4 Asymptotic Analysis of Bellman-Ford SSSP

This algorithm completes in $O(VE)$ time. The reason being that the initialization process takes $O(V)$ time, the relaxing process is completed E times for every vertex in V, and the final check runs E times. Before accounting for constants and unnecessary numbers, this would seem like $O(2V * 2E)$, although once simplified it becomes $O(VE)$.

3 Greedy Algorithms and Fractional Knapsack

Dynamic programming is a way of finding the best approach to solving a problem, and using the optimal solution. A very simple version of this is the greedy algorithm that can be used for the fractional knapsack problem. Say that there are multiple different spices, each in a different amount, and a different cost per unit. You want to fit as much value, in spice, as you can into your knapsack of finite capacity. The greedy algorithmic approach to this would have you do a defined set of steps.

1. Find the unit price of each of the spices by dividing their total price by their quantity.
2. Sort the spices in order from most expensive per unit to least expensive per unit.
3. Starting with the most expensive spice, take as much as you can of that spice. If not all will fit, take whatever fraction will fit in the knapsack. Continue until the knapsack is full.

3.1 `Greedy::greedyAlgo()`: Filling the Knapsacks

This method is called by the method which reads in the input file whenever a new knapsack capacity is given. This means that by this point in the processing,

all of the spices have been inputted, their individual unit prices have been calculated, and they have been sorted from least to greatest using Merge Sort (more on this later). With all of this set up, the stealing of the spices can commence.

```
void Greedy::greedyAlgo(int capacity)
{

    struct Portion
    {
        double qty;
        Spice *spice;
    };

    int cTotal = 0;
    vector<Portion *> ks;
    double earnings = 0;
    while (cTotal != capacity)
    {

        for (int i = spices.size() - 1; i > 0; i--)
        {
            if ((cTotal + spices[i]->qty) <= capacity)
            {

                Portion *p = new Portion();
                p->qty = spices[i]->qty;
                p->spice = spices[i];
                cTotal += spices[i]->qty;
                ks.push_back(p);
                earnings += p->spice->tPrice;
            }
            else if (((capacity - cTotal) > 0) && ((cTotal + spices[i]->qty) > capacity))
            {

                Portion *part = new Portion();
                part->qty = capacity - cTotal;
                part->spice = spices[i];
                cTotal += part->qty;
                ks.push_back(part);
                earnings += ((part->spice->uPrice) * (part->qty));
                break;
            }
        }
    }
    ...
    // printout code not integral to algorithm
    ...
}
```

Two things must be discussed before explaining the code above. First of all, there is the `struct` called `Portion` that was used to handle the data for

each portion of the spices which is defined to be used in this function. This structure will hold the quantity of that spice that is being taken, and the spice itself. Second, there is the prospective issue of not sorting from greatest to least. On line 16, it is seen that my iterator variable `i` is counting down from the last index of the spice list to the first. With this little tweak, my array of spices can be read in the same way, and with the same complexity as it is still using merge sort, and it is still iterating the same number of times only in reverse. Now, on to the process.

1. Start the current load counter `cTotal` and the `earnings` variables to 0, and initialize the vector to hold `Portion` structs and call it `ks`(lines 10-13).
2. Continue the following process until the `cTotal` variable matches the `capacity` of the knapsack, as passed in to the method (line 13).
3. Iterate over the list of sorted spices from greatest to smallest by iterating in reverse (line 16).
4. Check if the whole amount of the spice could fit in the knapsack (line 18)
    - If it can, add the whole spice as a portion to the `ks` vector with its quantity and the spice itself (lines 21-23, 25). Next update the `cTotal` with the new volume, and the total price of the spice (lines 24,26).
    - If it cannot, add an amount equivalent to the remaining space in the knapsack (lines 31-33, 35). Finally, update the `cTotal` and the `earnings` variables(34,36). Break out of the for loop (line 37).

3.2 Asymptotic Analysis of The Greedy Algorithm for Fractional Knapsack

Fractional knapsack runs in $O(n \lg n)$ time. The reason being that in the beginning, the process will just iterate until it cannot fit whole spice piles into its knapsack (n), but as it gets closer to filled there are fewer and fewer things that can be fit into the knapsack ($\lg n$).

4 Appendix

4.1 Bellman-Ford SSSP Extras

4.1.1 Header Files

GraphManager.h : The Graph Class

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include "Vertex.h"
#include "Edge.h"
using namespace std;
class GraphManager{
    public:
    vector<Vertex*> graph;
    vector<Edge *> weightMatrix;
    void fileReader(string fileName);
    void init(Vertex * source);
    void relax(Edge * edge, Vertex * from, Vertex * dest);
     bool Bellman();
     void printPath(Vertex * source, Vertex * dest);
};
```

- `graph` and `weightMatrix` hold the vertices and edges respectivley
- `fileReader()` is used to read the file and also run the algorithm
- `init()` is used to initialize the graph
- `relax()` is used to relax the edges of the graph
- `Bellman()` is the method used to carry out the process of the algorithm
- `printPath()` is used to print the paths from the source vertex to each of the other vertices

Edge.h : Edges of the Weighted Graph

```
#pragma once
#include <iostream>
#include "Vertex.h"
class Vertex;
class Edge
{
public:
    int weight;
    Vertex *dest;
    Vertex *from;
};
```

- `weight` holds the weight of the edge
- `*dest` holds a pointer to the destination vertex of the edge
- `*from` holds a pointer to the origin vertex of the edge

Vertex.h : Vertices of the Weighted Graph

```
#pragma once
#include <iostream>
#include <vector>
#include "Edge.h"
class Edge;
class Vertex
{
public:
    int id;
    int distance;
    Vertex *prev;
};
```

- `id` holds the id of the vertex so it can be identified later
- `distance` holds a the distance to a given vertex from the source vertex
- `*prev` holds a pointer to the vertex directly before

4.1.2 Helper Methods

`GraphManager::fileReader()` **: Reading in the Input File**

```
void GraphManager::fileReader(std::string fileName)
{

        fstream newfile;

        string ln;
        int count = 0;
        bool started = false;
        newfile.open(fileName, ios::in);
        if (newfile.is_open())
        {
                while (getline(newfile, ln))
                {

                        if (ln.find("new graph") != string::npos)
                        {

                                if (started)
                                {
cout << "**********************************"<<endl;
                                        Bellman();
```

```
22
23                                      weightMatrix.clear();
24                                      graph.clear(); //restarting the graph
25                              }
26                              started = true;
27                      }
28                      else if (ln.find("add vertex") != string::npos)
29                      {
30                              int id = stoi(ln.substr(11));
31                              Vertex *vertex = new Vertex();
32                              vertex->id = id;
33                              graph.push_back(vertex);
34                      }
35                      else if (ln.find("add edge") != string::npos)
36                      {
37
38                              int hyph = ln.find("-");
39
40                              int num1 = stoi(ln.substr((ln.find("edge") + 5), 2));
41                              int num2 = stoi(ln.substr(hyph + 2, 2));
42                              int weight = stoi(ln.substr(hyph + 4));
43
44                              for (int i = 0; i < graph.size(); i++)
45                              {
46                                      if (graph[i]->id == num1)
47                                      {
48                                              for (int j = 0; j < graph.size(); j++)
49                                              {
50                                                      if (graph[j]->id == num2)
51                                                      {
52
53                                                              Edge *e = new Edge();
54                                                              e->weight = weight;
55                                                              e->dest = graph[j];
56                                                              e->from = graph[i];
57
58                                                              weightMatrix.push_back(e);
59
60                                                      }
61                                              }
62                                      }
63                              }
64                      }
65              }
66      }
67      newfile.close();
68      cout << "*********************************"<<endl;
69      Bellman();
70      return;
71  };
```

1. If somewhere in the line it says ”add vertex”, assume this means it is defining a vertex. Grab the integer id from the assumed position in the string, and assign it to the `id` of a new vertex before adding it to the graph(lines 28-34)

2. If the line says ”add edge” pull out the ids of both of the defined vertices, and the weight (lines 35-42). Using two ”fingers” from nested for loops, find the two verticies in the graph, make them into the `from` and `dest` on a new edge, add the weight, and push the edge back on to the list of edges (lines 35-68)

3. If the line says new graph, we can assume that all of the information for the previous graph has been given. Although, if this is the first time seeing New Graph, this could just be the beginning of the file. So, if it is the first time simply raise the `started` flag and continue. Otherwise, run `Bellman()`, and then clear the graphs vertices and edges.(lines 15-27)

4. Since ”new graph” only occurs before a new graph, we need to run `Bellman()` once more at the end to process the final data points (line 68).

`GraphManager::printPath()` : Printing the Paths From The Source

```
1  void GraphManager::printPath(Vertex *source, Vertex *dest)
2  {
3          vector<Vertex *> traceback;
4          cout << source->id << "->" << dest->id << " cost is " << dest->distance << "; path: ";
5          bool found = false;
6          Vertex *currentV = dest;
7          while (!found)
8          {
9                  if (currentV->id != source->id)
10                 {
11
12                         traceback.push_back(currentV);
13                         currentV = currentV->prev;
14                 }
15                 else
16                 {
17
18                         found = true;
19                 }
20         }
21         cout << source->id;
22
23         for (int i = traceback.size() - 1; i >= 0; i--)
24         {
25                 cout << "->" << traceback[i]->id;
26         }
27         cout << endl;
28 }
```

1. Start by printing out the cost of the path by taking the `distance` of the `dest` vertex (line 4).

2. Make a placeholder `Vertex` pointer at the destination vertex that was passed to the method, and make a `vector` to hold the vertices that are found on the trace back through the path (lines 3, 6).

3. Starting with our placeholder at the destination, add our placeholder vertex to the traceback, and set the new placeholder to the previous vertex of our placeholder vertex (lines 12,13). Continue this process until the placeholder vertex `currentV` matches our `source` vertex (lines 7-20)

4. Starting from the back, print out the id of each of the vertices in the `traceback vector` (lines 23-27).

4.2 Fractional Knapsack Extras

4.2.1 Header Files

Greedy.h : The Knapsack Class

```
1  #pragma once
2  #include <string>
3  #include <vector>
4  #include "Spice.h"
5  using namespace std;
6  class Greedy
7  {
8  public:
9      void setupSpices(std::string fileName);
10     void greedyAlgo(int capacity);
11     vector<Spice *> spices;
12 };
```

- `setupSpices()` reads the input file and runs the algorithm
- `greedyAlgo()` runs a greedy algorithm for the Fractional Knapsack problem
- `spices` holds a list of spices

Spice.h : The Spice Data Structure

```
1  #pragma once
2  #include <string>
3
4  class Spice{
5      public:
6      std::string color;
7      double tPrice;
8      int qty;
9      int uPrice;
```

```
10
11
12      void setPrice(double tPrice, double qty){
13          this->tPrice = tPrice;
14          this->qty = qty;
15          this->uPrice = tPrice / qty;
16
17      }
18  };
```

- `color` is used to identify the spice
- `tPrice` holds the total price of the spice
- `qty` holds the quantity of the spice
- `uPrice` holds the unit price for the spice
- `setPrice()` takes both the total price and the quantity in, and sets the values for the total price, quantity, and the unit price by dividing the total price by the quantity

4.2.2 Helper Methods

`Greedy::setupSpices()` **: Setup for Spices and Running the Algorithm**

```
1   void Greedy::setupSpices(std::string fileName)
2   {
3
4       fstream newfile;
5
6       string ln;
7
8       bool sorted = false;
9       newfile.open(fileName, ios::in);
10      if (newfile.is_open())
11      {
12          while (getline(newfile, ln)) //read each line of the file
13          {
14
15              if (ln.find("knapsack capacity") != string::npos)
16              {
17
18                  if (!sorted)
19                  {
20                      //sort the stuff
21                      Sort::mergeSort(spices, 0, spices.size() - 1);
22                      sorted = true;
23                  }
24                  int cap = stoi(ln.substr(19, 3));
25
```

```
26                  cout << "**********************************" << endl;
27                  greedyAlgo(cap);
28                  cout << "**********************************" << endl;
29              }
30              else if (ln.find("spice name") != string::npos)
31              {
32                  vector<string> parts;
33
34                  for (int i = 0; i < ln.size(); i++)
35                  {
36                      if (ln[i] == ';')
37                      {
38                          for (int j = (i - 1); j > 0; j--)
39                          {
40                              if (ln[j] == '=')
41                              {
42
43                                  string sub = ln.substr(j + 2, (i - j) - 2);
44
45                                  parts.push_back(sub);
46
47                                  break;
48                              }
49                          }
50                      }
51                  }
52
53                  Spice *nSpice = new Spice();
54                  nSpice->color = parts[0];
55                  nSpice->setPrice(stod(parts[1]), stod(parts[2]));
56                  spices.push_back(nSpice);
57
58
59              }
60          }
61      }
62      newfile.close();
63
64      return;
65  };
```

1. If the current line contains ”spice name”, we can assume that this is the data for a spice. Because we know that the three data points for the spice always occur between an ”=” and a ”;”, and each of the three data points always occur in the same order, we can systematically find each point. We will keep these points in a string vector called `parts`.(line 30)

2. Whenever the outer for loop has its index on a semicolon(line 36), back track from this index until a ”=” is found (38-40).

3. Take the substring of the line using the two indexes `i` and `j`, adjusting slightly to mitigate white space. Then, add this substring to `parts`. Break the inner loop to continue searching for the next data point's semicolon(lines 43-47).

4. Make a new spice object, using the first item in `parts` as the color, and using the second and third items as the total price and quantity. This is applied using the `Spice::setPrice()` function which was described previously. Once created, add it to the list of spices (lines 53-56).

5. If the line contains "knapsack capacity", we can assume that this is giving the capacity of a knapsack. Therefore we know that all of the spices have been given and we can begin filling our knapsack. (line 15)

6. If this is the first time the algorithm is running, as denoted by the `started` flag not being set, then run merge sort on the spices to sort them from least expensive to most expensive by unit price. This sort is largely unchanged thanks to the ability to access items from a vector in a similar way to arrays. Other than changes to signatures, and changing to comparing the spices unit prices, nothing has been changed from previous uses of this sort.(lines 18-23)

7. After this process has been ran, or skipped, grab the correct substring of the line and turn it into an integer. This will be the capacity of the knapsack. Finally run the `greedyAlgo()` function with the correct capacity for the knapsack. (lines 24-27)

`Greedy::greedyAlgo()` **: The Output Functionality**

This section was added in the appendix as it was unneeded in the main portion of the document to convey the process of the algorithm. By this point, the knapsack is filled and all that is left is to print out what we escaped with.

```
1  void Greedy::greedyAlgo(int capacity)
2  {
3
4  ...
5  ...//Actual Algorithm Code
6  ...
7
8      cout << "Knapsack of capacity " << capacity << " is worth " << earnings << " quatloos and" << endl;
9      cout << "contains ";
10     for (int n = 0; n < ks.size(); n++)
11     {
12
13         if (n == 0)
14         {
15             if (ks[n]->qty == 1)
16             {
17                 cout << ks[n]->qty << " scoop of " << ks[n]->spice->color;
18             }
```

```
19              else
20              {
21                  cout << ks[n]->qty << " scoops of " << ks[n]->spice->color;
22              }
23          }
24          else
25          {
26              if (ks[n]->qty == 1)
27              {
28                  cout << ", " << ks[n]->qty << " scoop of " << ks[n]->spice->color;
29              }
30              else
31              {
32                  cout << ", " << ks[n]->qty << " scoops of " << ks[n]->spice->color;
33              }
34          }
35      }
36      cout << "." << endl;
37  }
```

1. Print out an opening line containing the total `earnings` from the knapsack (line 8)
2. For each `Portion` in the knapsack, list it's quantity followed by its color. Adjust grammar based on the quantity and whether or not it is the first item in the list. Meaning, adding an "s" to "scoop" for quantities greater than 1, and adding a comma before the statement if it is not the first item (lines 10-35)
3. Don't forget the period. (line 36)

4.3 Extras

Main.cpp : Putting Everything Together

```
1   #include <iostream>
2   #include <string>
3   #include "GraphManager.h"
4   #include "Greedy.h"
5   using namespace std;
6
7   int main()
8   {
9
10          GraphManager *gm = new GraphManager();
11          gm->fileReader("graphs2.txt");
12          Greedy * greed = new Greedy();
13          greed->setupSpices("spice.txt");
14          return 0;
15  }
```

1. Setup and run the `GraphManager` for Bellman-Ford SSSP with the ”graphs2.txt” file (lines 10,11)

2. Setup and run the `Greedy` object for fractional knapsack using the input file ”spice.txt” (lines 12,13)

References

8, Manan Soni May. “Bellman-Ford Algorithm in C and C++.” The Crazy Programmer, 3 June 2017, www.thecrazyprogrammer.com/2017/06/bellman-ford-algorithm-in-c-and-c.html.

4.4 Conclusion

Closing Thoughts

As with all of the assignments this semester, we were tasked with implementing algorithms. This meant that once we understood the algorithm well enough, we were generally able to complete the project without having to ”Solve” anything new. Other than some slight issues with the language that I ran into, or issues when it came to not completely understanding the algorithm.

For this reason, I enjoyed this assignment as I was able to solve somewhat open ended problems like how to use an existing sorting algorithm without doing extra unnecessary work. Or even the way I devised to parse the input strings for the file. To me, these were the most satisfying problems to solve.

as a wise person once said, ” give a person a program and they’ll be frustrated for a day, teach a person to program and they’ll be frustrated for the rest of their life.”. $\{S.A.\}$