Assignment Four: Bellman-Ford and Greedy Algorithms

Aidan Carr
Aidan.Carr1@Marist.edu

December 8, 2023

1 Graphs

1.1 What is a Directed Graph?

In Assignment 3, the concepts of undirected graphs were introduced. A directed Graph is, similarly, a collection of Vertices connected together with a series of edges. However, these Edges have a specified start and end Vertex. These Edges can have values appointed to them called weights. Below in Figure 1.1, each colored dot represents a Vertex and each arrow represents the directed Edge with a weight.

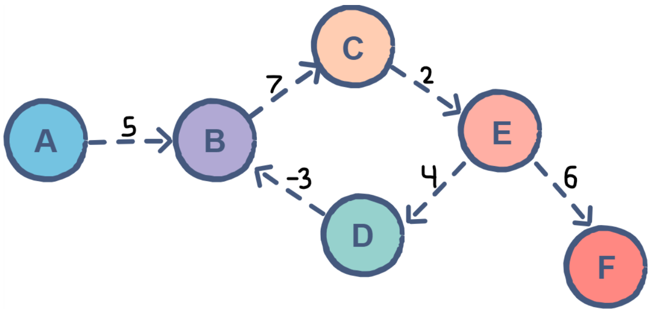


Figure 1.1: A Directed Graph[1]

Imagine this Graph represents a map of a city. Each Vertex is a location, each Edge represents a road with the flow of traffic and the time it takes to travel that road. GPS applications can use Graphs to calculate to quickest or shortest path from one point to another.

1.2 Assignment Goals

Similar to Assignment 3, the first step will be to read the file–graphs2.txt. This file is filled with a few commands to create four different Graphs. For each Graph, a series of Vertices and a series of Edges will be created. A major difference between this assignment and the previous is the existence of the Edges class. This eliminates to need for known neighbors of Vertices.

For each Graph, the goal of this assignment is to find the single source shortest paths (SSSP). This means finding the smallest weighted path from a start Vertex to every other Vertex. The Bellman-Ford algorithm will be used to find these shortest paths.

[1]Image: https://www.educative.io/answers/directed-graphs-vs-undirected-graphs.

1.3 Coding a Vertex Class

This Vertex class is a slight modification from Assignment 3's Vertex class.

```cpp
class Vertex {

public:
    string id;
    bool isProcessed;

    //for Bellman-Ford:
    int distance;
    Vertex* predecessor;

    //Constructor
    Vertex(string idInput){
        id = idInput;
        isProcessed = false;
    }
};
```

Figure 1.2: Vertex Class in C++

Check out Figure 1.2. Each Vertex object has an `id`, and an `isProcessed` variable. For the Bellman-Ford algorithm, `distance` will keep track of the total weight/cost/distance it has taken to get from the source Vertex to this one. The `predecessor` Vertex pointer will point to the previous Vertex in the path that has led to this one. These values will be filled in when the code runs through the Bellman-Ford algorithm.

1.4 Coding an Edge Class

```cpp
 1 class Edge {
 2 
 3 public:
 4     //this makes the Graph directed
 5     Vertex* from;
 6     Vertex* to;
 7     int weight;
 8 
 9     //Constructor
10     Edge(Vertex* fromInput, Vertex* toInput, int weightInput){
11         from = fromInput;
12         to = toInput;
13         weight = weightInput;
14     }
15 };
```

Figure 1.3: Edge Class in C++

The Edge class is a new Class that was not made in Assignment 3. Lines 4 through 7 of Figure 1.3 store the attributes for each Edge object. We have `from` and `to` Vector pointers, as well as a `weight` variable to store that. The constructor on line 10 uses inputs to appoint all three variables.

1.5 Coding a Graph Class

The Graph class's attributes on line 4 and 5 of Figure 1.4 are `vertices` and `edges`. Each is a vector of the respective object pointers. The constructor on line 8 does not appoint any value to these attributes. `findVertexById()` on line 13 takes a target string `id` as input and traverses the vector of Vertices to find the Vertex pointer. When found on line 17, the index is returned. If not, `-1` is returned on line 22.

The `isEmpty()` function returns true or false based on if the `vertices` vector is empty or not.

```
1  class Graph {
2  
3  public:
4      vector<Vertex*> vertices;
5      vector<Edge*> edges;
6  
7      //constructor
8      Graph(){
9      }
10 
11     //METHODS:
12     //return int location of Vertex in Graph given string id
13     int findVertexById(string target){
14 
15         //traverse the vector of Vertices
16         for (int i = 0; i < vertices.size(); i++){
17             if (isEqual(target, vertices[i]->id)){
18                 //return index
19                 return i;
20             }
21         }
22         return -1;
23     }
24 
25     //return true if no Vertices in vector
26     bool isEmpty(){
27         return vertices.empty();
28     }
```

Figure 1.4: Graph Class Methods in C++

Figure 1.5 below shows two more Graph methods. `addVertex()` on line 2 creates a Vertex object using the given `id` and adds it the the `vertiecs` vector.

On line 10, `addEdge()` is defined where it first checks if the indexes are valid Vertices on line 12. If so, an Edge is created with the given information, this is put into the `edges` vector.

```
1      //add Vertex
2      void addVertex(string id){
3          Vertex* myVertex = new Vertex(id);
4          //add new Vertex to the list
5          vertices.push_back(myVertex);
6      }
7  
8  
9      //add Edge
10     void addEdge(int index1, int index2, int weight){
11         //only add edge if the Vertices exist in vertices vector
12         if (index1 != -1 && index2 != -1){
13             Edge* myEdge = new Edge(vertices[index1], vertices[index2], weight);
14             edges.push_back(myEdge);
15         }
16     }
```

Figure 1.5: More Graph Class Methods in C++

The final Graph method in Figure 1.6 is the `reset()` function. This method deletes every Vertex in the `vertices` vector on lines 5 through 7 and deletes every Edge on lines 12 through 14. Both vectors all also cleared as well on lines 9 and 16.

```
1      //delete all Graph contents
2      void reset(){
3
4          //delete each Vertex
5          for (int i = 0; i < vertices.size(); i++){
6              delete vertices[i];
7          }
8          //reset the Vertices vector
9          vertices.clear();
10
11         //delete each Edge
12         for (int i = 0; i < edges.size(); i++){
13             delete edges[i];
14         }
15         //reset the Edges vector
16         edges.clear();
17     }
```

Figure 1.6: Reset the Graph Method in C++

The final two Graphs methods are `bellmanFord()` and `printBellmanFord()`. These will be covered later on.

1.6 Reading and Interpreting the File

In the main function of the program, the first step is to read the file into a vector. This process has been explained in Assignment 3. Looking at lines 12 through 15 of Figure 1.7, after the `fileCommands` vector has been created, the final element is removed, and the line `"new graph"` is added to the end. The `new graph` command will be an indicator that the graph is complete and ready for the Bellman-Ford process.

```
1      string currentLine;
2      if (graphsFile.is_open()){
3
4          //read the file into a vector
5          while (graphsFile){
6              //insert commands into vector
7              std::getline(graphsFile, currentLine);
8              fileCommands.push_back(currentLine);
9              fileLength++;
10         }
11
12         //IO duplicates final line, delete it
13         fileCommands.pop_back();
14         //print out final graph
15         fileCommands.push_back("new graph");
16     }
```

Figure 1.7: Reading the File in C++

```
-- CLRS and class example
new graph
add vertex 1
add vertex 2
add vertex 3
add vertex 4
add vertex 5
add edge 2 - 3  5
add edge 2 - 4  8
add edge 2 - 5 -4
add edge 3 - 2 -2
add edge 4 - 3 -3
add edge 4 - 5  9
add edge 5 - 3  7
add edge 5 - 1  2
add edge 1 - 2  6
add edge 1 - 4  7
```

Figure 1.8: Example Graph Commands

Figure 1.8 above shows a snippet of commands to be interpreted in the assignment. The difference here compared to Assignment 3 is the addition of the weight at the end of the `add edge` command. The other three commands (`-- comment`, `new graph`, and `add vertex`) remain the same.

Figure 1.9 represents the Graph that the commands build. Vertex `"1"` is the first vertex created, so it will be the source Vertex for SSSP.

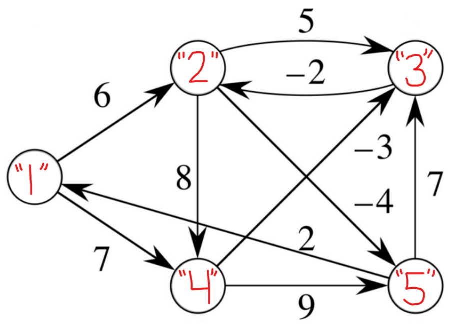


Figure 1.9: Example Graph Visualization[2]

When adding a Vertex, the string `id` is retrieved from the file and a new object is created with the `addVertex()` method in Figure 1.10.

```
        //add Vertex
        else if (currentLine.compare(4,6,"vertex") == 0){
            string id = currentLine.substr(11, 11-strLength);

            //add Vertex to Graph by id
            myGraph.addVertex(id);
        }
```

Figure 1.10: Add Vertex Command in C++

[2]Image: https://www.labouseur.com/courses/algorithms/Bellman-Ford.pdf.

Adding an Edge requires a bit more code. A lot of lines are used to find the exact placement of `id1`, `id2`, and `weight`. These are found using substrings and parsing on lines 4 through 15. The indexes for the `id`s are found on lines 18 and 19. Then, on line 22, the `addEdge()` method from earlier is called with our newly found data from this line.

```
 1         //add Edge
 2         else if (currentLine.compare(4,4,"edge") == 0){
 3 
 4             //find first id
 5             int dashIndex = currentLine.find("-");
 6             string id1 = currentLine.substr(9, dashIndex - 10);
 7 
 8             //find second id
 9             int id2strLength = currentLine.substr(dashIndex+2, strLength-dashIndex-2).
       find(" ");
10             string id2 = currentLine.substr(dashIndex+2, id2strLength);
11 
12             //find weight
13             int weightIndex = dashIndex+2+id2strLength;
14             string weightStr = currentLine.substr(weightIndex, strLength - weightIndex);
15             int weight = std::stof(weightStr);
16 
17             //find index of each id
18             int index1 = myGraph.findVertexById(id1);
19             int index2 = myGraph.findVertexById(id2);
20 
21             //add Edge in the graph using the Vertex indexes and weight
22             myGraph.addEdge(index1, index2, weight);
23         }
```

Figure 1.11: Add Edge Command in C++

The last command– `new graph`– does either one of two options. First, if this is the first Graph, `isEmpty()` returns true on line 5 and skips this whole sequence. If the Graph is populated, the Graph is processed. Bellman-Ford is called on line 8. If the algorithm produces a satisfactory result, it will be printed on lines 10 through 12 using the `printBellmanFord()` method. If not, line 15 prints out a failure message. The Graph is then reset on line 19, allowing the new Graph to be made.

```
 1         //new Graph
 2         else if (currentLine.compare(0,3,"new") == 0){
 3 
 4             //if there is a previous Graph, process it
 5             if (! myGraph.isEmpty()){
 6 
 7                 //process Graph
 8                 if (myGraph.bellmanFord(0)){
 9                     //print Bellman-Ford output
10                     std::cout << "Bellman-Ford SSSP:" << std::endl;
11                     myGraph.printBellmanFord(0);
12                     std::cout << std::endl;
13                 }
14                 else {
15                     std::cout << "Bellman-Ford SSSP not possible.\n" << std::endl;
16                 }
17 
18                 //reset Graph object (+ Vector objects and Edge objects)
19                 myGraph.reset();
20             }
```

Figure 1.12: New Graph Command in C++

1.7 The Bellman-Ford Algorithm

An explanation of the Bellman-Ford Algorithm. We let the algorithm know the source vector as input on line 2 in Figure 1.13. The first step in the algorithm is to set each Vertex's distance to `ALMOST_INFINTY`, a very large number. We also appoint each Vertex's `predecessor` to `nullptr` because currently we have not found any path, therefore no distance to any Vertex. And on line 11 the distance of the source Vector is set to 0 because no traveling is needed to stay in the same place.

```
 1  //Bellman-Ford Algorithm
 2  bool bellmanFord(int sourceVectorIndex){
 3  
 4      //INIT-SINGLE SOURCE
 5      //for every Vertex...
 6      for (int v = 0; v < vertices.size(); v++){
 7          vertices[v]->distance = ALMOST_INFINITY;
 8          vertices[v]->predecessor = nullptr;
 9      }
10      //distance from start to yourself = 0
11      vertices[sourceVectorIndex]->distance = 0;
12  
13      //loop through all Vertices
14      for (int i = 1; i < vertices.size(); i++){
15  
16          //loop through all Edges
17          for(int e = 0; e < edges.size(); e++){
18  
19              //variables for RELAX
20              Vertex* fromU = edges[e]->from;
21              Vertex* toV = edges[e]->to;
22              int weight = edges[e]->weight;
23  
24              //RELAX
25              //if previously defined distance > new found distance, appoint new path
26              if (toV->distance > fromU->distance + weight){
27                  toV->distance = fromU->distance + weight;
28                  toV->predecessor = fromU;
29              }
30          }
31      }
32  
33      //loop through Edges to check for negative weight cycles
34      for(int e = 0; e < edges.size(); e++){
35  
36          //variables for negative weight cycle check
37          Vertex* fromU = edges[e]->from;
38          Vertex* toV = edges[e]->to;
39          int weight = edges[e]->weight;
40  
41          if (toV->distance > fromU->distance + weight){
42              return false;
43          }
44      }
45      //Bellman-Ford complete!
46      return true;
47  }
```

Figure 1.13: Bellman-Ford Algorithm in C++

Line 14 of Figure 1.13 begins a loop of all Vertices (except the source) and line 17 loops through all of the Edges for each Vector. This is the RELAX section of the algorithm where we search for shorter paths. Line 26 is crucial; here, we check if the distance of the `to` Vertex in this particular Edge is greater than the distance of the `from` Vertex plus the Edge `weight`. The algorithm is checking if the previously found path is longer than the new path. If a shorter path is found, lines 27 and 28 are executed, updating the `to` Vertex's `distance` and `predecessor`.

Once all the Vertices have been checked, the last step is to check for negative weight cycles. A negative weight cycle occurs when a Vertex manages to use Edges to loop back to itself with a negative distance. Graphs with negative weight cycles could then have paths of infinitely negative size. Line 41 checks if each

Vertex's distance is greater than each Edge's `from` Vertex pointing to it, plus the weight. `bellmanFord()` returns false if just one negative weight cycle is found.

Some people may say: "B- b- but, I love negative weight cycles!" But it is important to remember the effects of negative weight cycles in a SSSP Graph. Check out Figure 1.14.

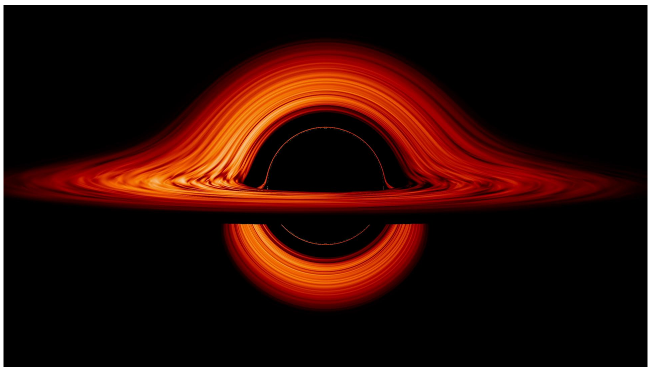


Figure 1.14: The Outcome of a Graph with Negative Weight Cycles[3]

1.8 Printing Bellman-Ford

The Bellman-Ford Algorithm modifies the Graph's Vertices and returns true or false if the algorithm is possible. To print the findings, a new Graph method will be created: `printBellmanFord()`. Check out Figure 1.15. On line 3, all Vertices are looped through (excluding the source Vertex). Within each loop, the path start, finish, and distance are all printed on lines 4 and 5.

```
1      //print Bellman-Ford
2      void printBellmanFord(int startIndex){
3          for (int endIndex = 1; endIndex < vertices.size(); endIndex ++){
4              std::cout << vertices[startIndex]->id << " -> " << vertices[endIndex]->id;
5              std::cout << " cost is " << vertices[endIndex]->distance << "; path: ";
6
7              //name this path and push it to a Stack
8              Vertex* pathNode = vertices[endIndex];
9              Stack* path = new Stack();
10
11             //traverse predecessors, push to Stack until the start is reached
12             while (pathNode != nullptr){
13                 path->push(pathNode);
14                 pathNode = pathNode->predecessor;
15             }
16
17             //pop and print out the Stack
18             path->pop();
19             while (! path->isEmpty()){
20                 std::cout << " -> ";
21                 path->pop();
22             }
23             std::cout << std::endl;
24         }
25     }
```

Figure 1.15: Printing Bellman-Ford Method in C++

A `pathNode` and Stack are created on lines 8 and 9. This is because we will move along the path backwards (`to` back to `from`), and a Stack will help reverse this order. Looping on line 12 until we reach the source Vertex, we keep pushing `pathNode`s. Once all path points have been added, on lines 18 through 23, the path is popped in reverse order, printing out the path from the source to the particular Vertex. To read more about Stacks, check out Assignment 1. Or, Google it or something.

[3]Image: https://universe.nasa.gov/black-holes/anatomy/.

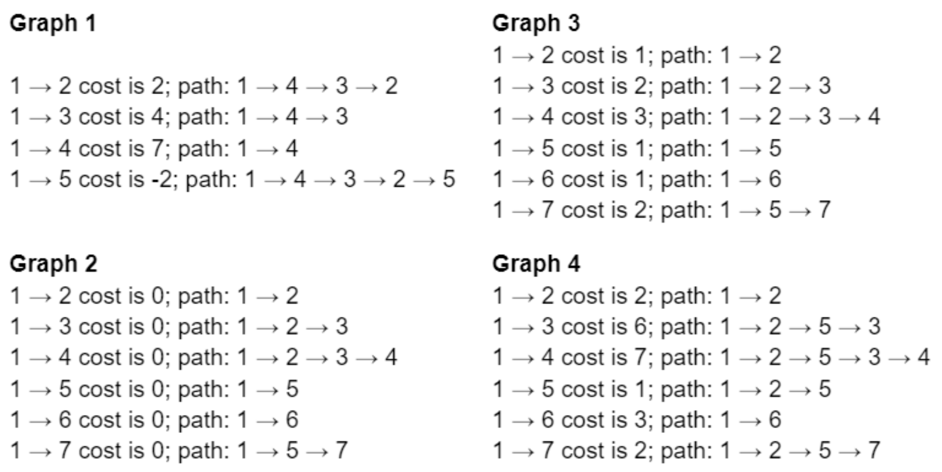
1.9 Bellman-Ford Complexity and Results

The complexity of Bellman-Ford is easy to calculate. Referring back to the Bellman-Ford algorithm in Figure 1.13, there is a nested for loop. The outer loop on line 14 repeats for every Vertex in the Graph, and the inner loop on line 17 loops through all the Edges.

This section of the algorithm is repeated for the number of Vertices multiplied by the number of Edges. Therefore, the time complexity can be represented as:

$$O(|V| * |E|)$$

The Bellman-Ford outputs of the Graphs created in `graphs2.txt`, are shown in Figure 1.16 below. Graph 1 represents the the SSSPs from the Graph in Figure 1.9.



```
Graph 1

1 → 2 cost is 2; path: 1 → 4 → 3 → 2
1 → 3 cost is 4; path: 1 → 4 → 3
1 → 4 cost is 7; path: 1 → 4
1 → 5 cost is -2; path: 1 → 4 → 3 → 2 → 5

Graph 2
1 → 2 cost is 0; path: 1 → 2
1 → 3 cost is 0; path: 1 → 2 → 3
1 → 4 cost is 0; path: 1 → 2 → 3 → 4
1 → 5 cost is 0; path: 1 → 5
1 → 6 cost is 0; path: 1 → 6
1 → 7 cost is 0; path: 1 → 5 → 7

Graph 3
1 → 2 cost is 1; path: 1 → 2
1 → 3 cost is 2; path: 1 → 2 → 3
1 → 4 cost is 3; path: 1 → 2 → 3 → 4
1 → 5 cost is 1; path: 1 → 5
1 → 6 cost is 1; path: 1 → 6
1 → 7 cost is 2; path: 1 → 5 → 7

Graph 4
1 → 2 cost is 2; path: 1 → 2
1 → 3 cost is 6; path: 1 → 2 → 5 → 3
1 → 4 cost is 7; path: 1 → 2 → 5 → 3 → 4
1 → 5 cost is 1; path: 1 → 2 → 5
1 → 6 cost is 3; path: 1 → 6
1 → 7 cost is 2; path: 1 → 2 → 5 → 7
```

Figure 1.16: Bellman-Ford Outputs[4]

2 Greedy Algorithms

2.1 The Fractional Knapsack Problem

Imagine you have a knapsack the can only hold a certain volume. Bringing this knapsack to a heist will allow you to fill it to the brim and leave. But how can you leave with the most value?

This section of the assignment attempts to find the best options for different knapsack sizes. The items we are stealing are scoops of differently valued spices. The method of taking the most value is to take all or as much as possible of the most valuable spice. Then, the next most valuable spice is scooped, then the next... until the sack is full.

Figure 2.1: A Generous Scoop of Spice, Dune (2021)[5]

[5]Image: https://news.cnrs.fr/articles/dissecting-the-spice-of-dune.

2.2 Coding a Spice Class

This simple Spice class in Figure 2.2 has some attributes with a constructor. The constructor on line 11 sets a `name`, `totalPrice`, and `quantity` from the parameters. From this information, a fourth variable `unitPrice` is calculated on line 15. This number will be used to determine the particular Spice's value. A higher unit price will be more valuable for the heist.

```
 1 //SPICE CLASS
 2 class Spice {
 3 
 4 public:
 5     string name;
 6     float totalPrice;
 7     int quantity;
 8     float unitPrice;
 9 
10     //Constructor
11     Spice(string nameInput, float priceInput, int qtyInput){
12         name = nameInput;
13         totalPrice = priceInput;
14         quantity = qtyInput;
15         unitPrice = totalPrice / quantity;
16     }
17 };
```

Figure 2.2: Spice Class in C++

2.3 Reading the Spice File

The assignment requires us to read a `spice.txt` file, shown in Figure 2.3. There are only two types of command lines. One defines a new Spice object (lines 4 through 7), and one defines a knapsack capacity (lines 10 through 14).

```
 1 -- She who controls the spice controls the universe.
 2 
 3 -- Available spice to take
 4 spice name = red;    total_price =  4.0;  qty = 4;
 5 spice name = green;  total_price = 12.0;  qty = 6;
 6 spice name = blue;   total_price = 40.0;  qty = 8;
 7 spice name = orange; total_price = 18.0;  qty = 2;
 8 
 9 -- Available knapsacks in which to keep spice
10 knapsack capacity =  1;
11 knapsack capacity =  6;
12 knapsack capacity = 10;
13 knapsack capacity = 20;
14 knapsack capacity = 21;
```

Figure 2.3: Spice Text File

Just like earlier in this assignment, and in Assignment 3, the lines of the text file will be read into a vector. Check out previous assignments for mode detailed code listings.

Within the main file, Figure 2.4 shows when two vectors are created on lines 2 and 3. `knapsackCapacites` will be used to store the integer values of the knapsacks' capacities. `spices` will store all the Spices in one place.

Line 7 loops through all lines of the text file. Lines 13 through 19 deal with the comment command and empty lines.

```
1      //create vectors for all information
2      vector<int> knapsackCapacities;
3      vector<Spice*> spices;
4
5
6      //INTERPRET ALL COMMANDS
7      for (int line = 0; line < fileLength; line ++){
8
9          //get line as a string, and find length
10         string currentLine = fileCommands[line];
11         int strLength = currentLine.length();
12
13         //empty line, ignore line
14         if (currentLine.compare(0,1,"") == 0){
15         }
16
17         //comment, ignore line
18         else if (currentLine.compare(0,2,"--") == 0){
19         }
```

Figure 2.4: Looping the Spice Commands in C++

Still inside the for loop in Figure 2.5, the code checks if the current command is a new Spice command. If so, lines 4 through 7 uses substrings and indexing to get the `name` of the Spice. Lines 9 through 14 gets the `totalPrice` using a similar method. `quantity` is found on lines 16 through 21, then the Spice object is created using the variables found from the text file on line 24. This object is put into the `spices` vector for safe keeping.

```
1      //new Spice
2      else if (currentLine.compare(0,5,"spice") == 0){
3
4          //get name
5          int equalsIndex = currentLine.find("=");
6          int colonIndex = currentLine.find(";");
7          string name = currentLine.substr(equalsIndex +2, colonIndex - equalsIndex-2);
8
9          //get total_price
10         currentLine = currentLine.substr(colonIndex +2, strLength - colonIndex-2);
11         equalsIndex = currentLine.find("=");
12         colonIndex = currentLine.find(";");
13         string strPrice = currentLine.substr(equalsIndex+2, colonIndex - equalsIndex-2);
14         float totalPrice = std::stof(strPrice);
15
16         //get qty
17         currentLine = currentLine.substr(colonIndex +2, strLength - colonIndex-2);
18         equalsIndex = currentLine.find("=");
19         colonIndex = currentLine.find(";");
20         string strQuantity = currentLine.substr(equalsIndex+2,colonIndex-equalsIndex-2);
21         int quantity = std::stof(strQuantity);
22
23         //create Spice object using data
24         Spice* mySpice = new Spice(name, totalPrice, quantity);
25         spices.push_back(mySpice);
26     }
```

Figure 2.5: Reading a New Spice Command in C++

Next, we will will check if the command is a knapsack capacity. If so, lines 3 through 5 find the integer value for this new `capacity`, then put it in the `knapsackCapacities` vector for safe keeping on line 7.

```
1      //new knapsack size
2      else if (currentLine.compare(0,8,"knapsack") == 0){
3          int equalsIndex = currentLine.find("=");
4          string strCapacity = currentLine.substr(equalsIndex+2, strLength-equalsIndex-3);
5          int capacity = std::stof(strCapacity);
6          //add capacity to vector of knapsacks
7          knapsackCapacities.push_back(capacity);
8      }
```

Figure 2.6: Reading a Knapsack Command in C++

2.4 Coding a Greedy Algorithm

Once the loop through commands is completed, we must sort the `spices` vector in descending order of unit price. This makes the first Spice the most valuable, and the last the least valuable. After calling `spices = sortValues(spices);` in the main function, the program comes to `sortValues()` in Figure 2.7. This is a modified Selection Sort code. `maxPosition` is found in every loop, then swapped on lines 15 through 17 with the current Spice.

```
1  //Return vector of Spices in unit price descending order (most->least valuable)
2  vector<Spice*> sortValues(vector<Spice*> spices){
3      int maxPosition;
4
5      for (int i = 0; i < spices.size() - 1; i++){
6          maxPosition = i;
7          for (int j = i+1; j < spices.size(); j++){
8
9              //compare for better unit price
10             if (spices[j]->unitPrice > spices[maxPosition]->unitPrice){
11                 maxPosition = j;
12             }
13         }
14         //swap
15         Spice* temporarySpice = spices[maxPosition];
16         spices[maxPosition] = spices[i];
17         spices[i] = temporarySpice;
18     }
19     return spices;
20 }
```

Figure 2.7: Sort Function in C++

Back to the main program in Figure 2.8, we begin preparing for the greedy algorithm by looping through all of the knapsack capacities on line 2. Then a few variables are initialized. `capacity` is capacity of the current knapsack, `spiceNumber` is a counter to be incremented, `sackQuantity` is the current number of scoops in the knapsack, and `scoopDetails` will be an ongoing sentence for output purposes.

```
1      //loop through all knapsack
2      for (int sackNumber = 0; sackNumber < knapsackCapacities.size(); sackNumber++){
3
4          //knapsack variables
5          int capacity = knapsackCapacities[sackNumber];
6          int spiceNumber = 0;
7          int sackQuantity = 0;
8          float sackPrice = 0.0;
9          bool isFull = false;
10         string scoopDetails = "";
```

Figure 2.8: Preparing for the Greedy Algorithm in C++

Figure 2.9 shows the greedy algorithm. Using a while loop on line 2, we check the the knapsack has space and we still have spaces left to steal. Remember we are looping through the spices in value order. If there is enough room to take all of the Spice, execute lines 6 through 10. Here, `scoops`, `sackQuantity`, and `sackPrice` are all updated.

If we can only take a portion of Spice, execute lines 13 through 17. Here, the sack `scoops` and `sackQuantity` are maxed out. On line 16, the `sackPrice` is updated to add the `unitPrice` multiplied by `scoops`.

```
1          //loop through spices (in most valuable order) until sack is full
2          while (!isFull && spiceNumber < spices.size()){
3              int scoops;
4  
5              //if there is enough space, take all of the spice
6              if (spices[spiceNumber]->quantity <= capacity){
7                  scoops = spices[spiceNumber]->quantity;
8                  sackQuantity += scoops;
9                  sackPrice += spices[spiceNumber]->totalPrice;
10             }
11 
12             //if not, take only what is availible
13             else{
14                 scoops = capacity - sackQuantity;
15                 sackQuantity += scoops;
16                 sackPrice += spices[spiceNumber]->unitPrice * scoops;
17             }
18 
19             //check if sack is full (or at final spice)
20             if(sackQuantity == capacity || spiceNumber+1 == spices.size()){
21                 isFull = true;
22             }
23 
24             //gather all scoop details for printing
25             editPrintDetails(scoopDetails, scoops, spices, spiceNumber, isFull);
26 
27             //next Spice!
28             spiceNumber ++;
29         }
```

Figure 2.9: Greedy Spice Heist Algorithm in C++

Then the algorithm reevaluates the `isFull` variable in case the sack is now full on line 20. The function `editPrintDetails()` is called on line 25. This function will create and edit a string that will be printed out at the end of heist. Finally, the `spiceNumber` is incremented on line 28, moving on to the next Spice.

Below, in Figure 2.10, `editPrintDetails()` adds details about the most recent Spice taken onto the ever-changing `scoopDetails` string. This function checks if "scoops" should be plural on lines 4 through 9, and if there are more scoops to come on lines 11 and 12.

```
1  //create or add to a string of details that will be printed in the end
2  void editPrintDetails(string scoopDetails, int scoops, vector<Spice*> spices, int
       spiceNumber, bool isFull){
3      scoopDetails.append(to_string(scoops));
4      if (scoops == 1){
5          scoopDetails.append(" scoop of ");
6      }
7      else {
8          scoopDetails.append(" scoops of ");
9      }
10     scoopDetails.append(spices[spiceNumber]->name);
11     if (!isFull){
12         scoopDetails.append(", ");
13     }
14 }
```

Figure 2.10: Print Details Function in C++

Looking at the greedy algorithm in Figure 2.9, there is only one while loop. This loops through the number of Spices, in order to pick multiple Spices for our knapsack. The complexity of this section of the greedy algorithm is:

$$O(n)$$

This is because we loop through everything once. But, this is not the entire story. Before the greedy algorithm, the `spices` vector had to be sorted by `unitPrice`. Because of this, the time complexity of greedy algorithm is bounded by:

$$O(n^2)$$

Because there is a nested for loop in the Selection Sort algorithm, this overpowers the smaller complexity of the greedy section. If a less time inducing sort was used (like Merge Sort), the complexity of the whole greedy algorithm would again be equal to the sorting method's complexity.

So, the greedy algorithm of the Spice heist is determined by the complexity of the Spice value sorting method.

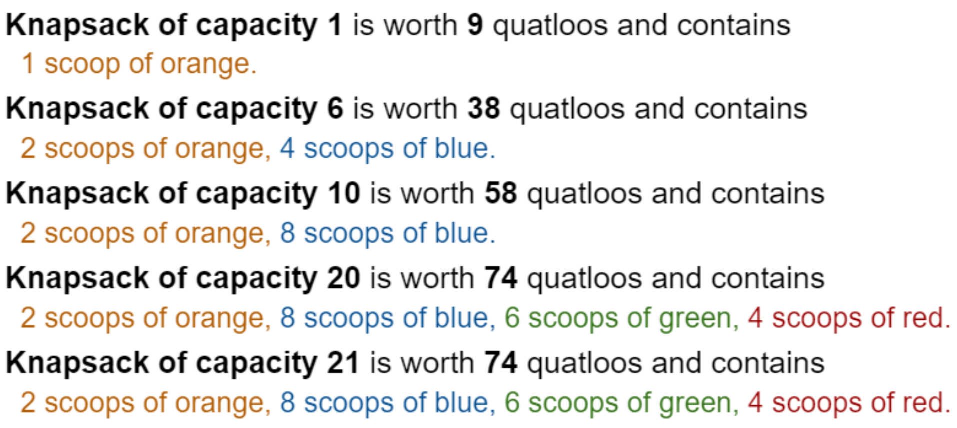
2.5 The Results of a Greedy Spice Heist

```
1        //print out capacity, price, and scoop details
2        std::cout << "Knapsack of capacity " << capacity;
3        std::cout << " is worth " << sackPrice << " quatloos";
4        std::cout << " and contains " << scoopDetails << ".\n" << std::endl;
```

Figure 2.11: Printing Out the Results in C++

One final piece of code can be found in Figure 2.11, at the end of the main function. Each knapsack capacity and price is printed, followed by the sack's details on line 4.

The output of `spice.txt` will produce the output found in Figure 2.12.



```
Knapsack of capacity 1 is worth 9 quatloos and contains
 1 scoop of orange.
Knapsack of capacity 6 is worth 38 quatloos and contains
 2 scoops of orange, 4 scoops of blue.
Knapsack of capacity 10 is worth 58 quatloos and contains
 2 scoops of orange, 8 scoops of blue.
Knapsack of capacity 20 is worth 74 quatloos and contains
 2 scoops of orange, 8 scoops of blue, 6 scoops of green, 4 scoops of red.
Knapsack of capacity 21 is worth 74 quatloos and contains
 2 scoops of orange, 8 scoops of blue, 6 scoops of green, 4 scoops of red.
```

Figure 2.12: Greedy Algorithm Outputs[6]