Assignment Five

Veronica Longley

December 10, 2021

Our Goal: In this assignment we will read a given file and create directed, weighted graphs based on the listed instructions. We will then find the Single Source Shortest Path (SSSP) from the first vertex to the other vertices. Next we will conduct a spice heist where we will attempt to fill knapsacks of various capacities with the most valuable spices so we are getting the most quatloos possible out of our heist.

1 Vertex Class

```
public class Vertex {
        //vertices are labelled with an id
        //dist and prev aid with SSSP
        private int id;
        private int dist;
        private Vertex prev;

        public Vertex()
        {
                id = 0;
                prev = null;
                dist = -1;
        }//Vertex

        public Vertex(int newid)
        {
                id = newid;
        }//Vertex

        public void setPrev(Vertex newPrev)
        {
                prev = newPrev;
        }//setPrev

        public Vertex getPrev()
        {
                return prev;
        }//getPrev

        public void setDist(int newDist)
        {
                dist = newDist;
        }//setDist

        //getter for id
        public int getid()
        {
                return id;
        }//getid

        public int getDist()
        {
                return dist;
        }//getDist

}//Vertex
```

First we must look at the vertex class. The vertices are what make up the graph and allow edges to be formed. Vertices are labelled with an integer id. The integer dist and Vertex prev are used in later classes to find the SSSP. This class is quite simple; we initialize and write getters and setters.

2 Edge Class

```
1
2  public class Edge {
3
4          public Vertex from;
5          public Vertex to;
6          private int weight;
7
8          public Edge()
9          {
10                 from = new Vertex();
11                 to = new Vertex();
12                 weight = 0;
13         }//Edge
14
15         public Edge(Vertex newFrom, Vertex newTo, int newWeight)
16         {
17                 from = newFrom;
18                 to = newTo;
19                 weight = newWeight;
20         }//Edge
21
22         public int getToID()
23         {
24                 return to.getid();
25         }//getToID
26
27         public int getFromID()
28         {
29                 return from.getid();
30         }//getFromID
31
32         public Vertex getTo()
33         {
34                 return to;
35         }//getTo
36
37         public Vertex getFrom()
38         {
39                 return from;
40         }//getFrom
41
42         public int getWeight()
43         {
44                 return weight;
45         }//get weight
46
47         public int getFromDist()
48         {
49                 return from.getDist();
```

```
50          }//getFromDistance
51
52          public int getToDistance()
53          {
54                  return to.getDist();
55          }//getFromDistance
56
57          public void detailsE()
58          {
59                  System.out.println(from.getid() + "  " + to.getid() + " " + weight);
60          }//detailsE
61  }
```

The edge class works similarly to the vertex class. An edge is made up of two vertices, from and to, and a weight. The weight indicates the 'cost' to travel that edge. When finding the SSSP we want the smallest 'cost' possible, and we need to make sure we are traveling the correct direction, so this class will take care of that. As before, we initialize, provide getters and setters and the last method (line 57) which will print details about the edge. We will use this to print a summary of the graph.

3 Graph Class

```
1
2   import java.util.ArrayList;
3   public class Graph {
4
5           private int numOfVertices;
6           public ArrayList<Vertex> theVertices;
7           public ArrayList<Edge> theEdges;
8
9           public Graph()
10          {
11                  theVertices = new ArrayList<>();
12                  theEdges = new ArrayList<>();
13                  numOfVertices = 0;
14
15          }//Graph
16
17          public void addVertex(int num)
18          {
19                  Vertex newVertex = new Vertex(num);
20                  theVertices.add(newVertex);
21                  numOfVertices++;
22                  //System.out.println("Added " + num);
23          }//addVertex
24
25          public void addEdge(int sourceID, int destID, int weight)
26          {
27                  Edge newEdge = new Edge(theVertices.get(sourceID-1),
28                  theVertices.get(destID-1), weight);
29                  //System.out.println("Added " + sourceID + " - " + destID + " "
30                  + weight);
31                  theEdges.add(newEdge);
32          }//addEdge
33
34          public ArrayList<Vertex> getVertices()
```

```
35          {
36                  return theVertices;
37          }//getVertices
38
39          public ArrayList<Edge> getEdges()
40          {
41                  return theEdges;
42          }//getEdges
43
44          public int getnumOfVertices()
45          {
46                  return numOfVertices;
47          }//getnumOfVertices
48
49          public void detailsG()
50          {
51                  for (int i = 0; i < theEdges.size();  i++)
52                  {
53                          theEdges.get(i).detailsE();
54                  }//for
55          }//detailsG
56  }//Graph
```

The graph class contains two array lists: one for the graph's vertices and one for the graph's edges. We also have a counter for the number of vertices, but that is not needed for this specific implementation. After initializing all of these, we have an addVertex() method (Line 17) where we create a new vertex, add it to the array list of vertices, and increment the number of vertices. Similarly for add edge, we create a new edge (adjusting for zero based indices) and add it to the array list of edges. We then have getters for both array lists and the vertices counter (line 34 - 47). Lastly we have a method that will call the previously mentioned detailsE() from the Edge class for each of the graph's edges so we may print a summary.

4 SSSP Class

```
1   import java.util.ArrayList;
2   import java.util.Stack;
3   public class SSSP
4   {
5           public SSSP(Graph graph)
6           {
7                   algorithm(graph);
8           }//SSSP
9
10          public void algorithm(Graph graph)
11          {
12                  int start = 0;
13                  ArrayList<Vertex> theVertices = graph.getVertices();
14                  ArrayList<Edge> theEdges = graph.getEdges();
15
16                  //initialize
17                  for(int i = 0; i <  theVertices.size(); i++)
18                  {
19                          theVertices.get(i).setDist(10000);
20                          theVertices.get(i).setPrev(null);
```

```
                }//for

                theVertices.get(start).setDist(0);

                //relax
                for(int j = 1; j<theVertices.size(); j++)
                {
                        for(int k = 0; k<theEdges.size(); k++)
                        {
                                relax(theEdges.get(k).getFrom(),
                                theEdges.get(k).getTo(),theEdges.get(k).getWeight());
                        }//for
                }//for

                //check for negative cycle
                for(int i = 0; i < theEdges.size(); i++)
                {
                        if(theEdges.get(i).getTo().getDist() >
                        (theEdges.get(i).getFrom().getDist() +
                        theEdges.get(i).getWeight()))
                        {
                                System.out.println("Negative Cycle");
                        }//if
                }//for

                printPath(theVertices);

        }//algorithm

        public void relax(Vertex from, Vertex to, int weight)
        {
                if(to.getDist() > (from.getDist()+ weight))
                {
                        to.setDist(from.getDist()+weight);
                        to.setPrev(from);
                }//if
        }//relax

        public void printPath(ArrayList<Vertex> vertices)
        {
                Stack<Integer> path = new Stack<>();
                System.out.println("Shortest Path: ");
                for(int i = 1; i < vertices.size(); i++)
                {
                        Vertex vertex = vertices.get(i);
                        System.out.print("1 --> " + vertices.get(i).getid() +
                        " cost is "+ vertices.get(i).getDist() + "; path is ");
                        while(vertex.getPrev() != null)
                        {
                                vertex = vertex.getPrev();
                                path.push(vertex.getid());
                        }//while
                        while(!path.isEmpty() )
                        {
                                int id = path.pop();
                                System.out.print(id + " --> ");
                        }//while
                        System.out.print(vertices.get(i).getid() + "\n");
```

```
79                  }//for
80          }//printPath
81  }//SSSP
```

The SSSP class is called in main to determine the path from Vertex 1 to every other vertex resulting in the lowest 'cost'. First we create two new array lists and use the getters to copy the vertices and edges of the graph we are looking at. We start by initializing dist to 10,000('infinite'). This does not need to be specific, but it needs to be a rather large number so when we check the first path it is in fact less cost wise than 10,000 and dist gets updated to that. The vertex that we are using as our 'source', the first vertex, gets set to have a distance of zero so we are starting our paths out with nothing and accumulating 'costs' as we go. We then call the relax function on each of the edges. Relax checks if the destination vertex distance is greater than the source vertex distance plus the weight of the edge between them. If it is we have found a shorter path and we can set the destination distance to be the source vertex distance plus the weight of the edge. Once this has been completed for each edge, we check if the graph has a negative cycle. A negative cycle occurs when each time your path goes through a loop in the graph the cost gets lower and lower. In this case, the SSSP is never found because the algorithm wants to continue to get the cost to be lower and never stops the negative cycle. Lastly, we will call the print path method for all of the vertices. PrintPath() (line 59) works by going through all of the vertices except the first and printing the SSSP from Vertex 1 to that vertex. We first push all of the prev Verticies onto a stack to reverse the order, then we pop them off one by one and print them out to show the SSSP from Vertex 1 to each other vertex.

5 ...In Main Method

```
1   public class main {
2
3           public static void main(String[] args)
4           {
5
6                   String fileName = "graphs2.txt";
7                   try
8                   {
9                   Scanner readFile = new Scanner (new File (fileName));
10                  String textLine = "";
11                  Graph newGraph = new Graph();
12                  String[] vertices;
13          String[] edges;
14          int graph = 0;
15          System.out.println("Directed Weighted Graphs: ");
16                  //Read instructions .txt file
17                  //make graphs line by line as opposed to storing file
18                  while (readFile.hasNextLine())
19                     {
20                          textLine = readFile.nextLine();
21                          if (textLine.equals("new graph"))
22                          {
23                                  newGraph = new Graph();
24                                  graph++;
25                          }//if
26
```

```
                        //if the line is not empty and is not a comment
                        else if (!(textLine.equals(""))&&(textLine.charAt(0)!='-'))
                        {
                                //if it contains 'vertex' we are adding a vertex
                                if(textLine.contains("vertex"))
                                {
                                        vertices = textLine.split("add vertex ");
                                        newGraph.addVertex(Integer.parseInt
                                        (vertices[1]));
                                }//if

                                //if the line contains edge we are adding an edge
                                else if (textLine.contains("edge"))
                                {
                                        edges = textLine.split("\\s+");
                                        newGraph.addEdge(Integer.parseInt(edges[2]),
                                        Integer.parseInt(edges[4]),
                                        Integer.parseInt(edges[5]));
                                }//if
                        }//else if

                        //if line is empty we are outputting SSSP
                        else if (textLine.equals("") )
                        {
                                System.out.println("Graph #"+ graph + ": ");
                                newGraph.detailsG();
                                SSSP nextSssp = new SSSP(newGraph);
                                System.out.println();
                        }//else if
                   }//while
                }//try

        catch(FileNotFoundException ex)
                {
                System.out.println("Failed to find file: "+ fileName);
                }//catch
        catch(InputMismatchException ex)
                {
                System.out.println("Type mismatch");
                System.out.println(ex.getMessage());
                }//catch
        catch(NumberFormatException ex)
                {
                System.out.println("Failed to convert String into an integer. ");
                System.out.println(ex.getMessage());
                }//catch
        catch(NullPointerException ex)
                {
                System.out.println("Null pointer exception. ");
                System.out.println(ex.getMessage());
                }//catch
        catch(Exception ex)
                {
                System.out.println("Oops, something went wrong. ");
                ex.printStackTrace();
                }//catch
        }//main
}//main
```

In main we will read through the file constructing the graph line by line as opposed to storing the instructions. While the file has next line we will read it, if the line contains new graph we will create a new graph. If the line is not empty and not a comment we will determine if we are creating a vertex or an edge. If we are creating a vertex we will remove the letters from the line, parse the integer and create a new vertex for the graph. If the line contains edge, we will split on spaces, parse each of the integers for vertices' ids and the weight of the edge, and finally add the edge. If the line is empty we know we are done describing the graph and we can print out a label for the graph, print out the details of the graph's edges, and call SSSP to find the Single Source Shortest Path from vertex one to every other vertex. This completes the section on directed weighted graphs.

Moving on to the Spice Heist...

6 Spice Class

```
1
2  public class Spice {
3
4          private String name;
5          private double perUnitValue;
6          private int quant;
7          private Spice next;
8
9          public Spice()
10         {
11                 name = null;
12                 perUnitValue = 0;
13                 quant = 0;
14                 next = null;
15         }//Spice
16
17         public Spice(String newName, double newPerUnitValue, int newQuant)
18         {
19                 name = newName;
20                 perUnitValue = newPerUnitValue;
21                 quant = newQuant;
22                 next = null;
23         }//Spice
24
25         public void setNext(Spice newNext)
26         {
27                 next = newNext;
28         }//setNext
29
30         public double getPerUnitValue()
31         {
32                 return perUnitValue/quant;
33         }//getUnitValue
34
35         public int getQuant()
36         {
37                 return quant;
38         }//getQuant
39
```

```
40          public String getName()
41          {
42                  return name;
43          }//getName
44
45          public Spice getNext()
46          {
47                  return next;
48          }//getNext
49  }//Spice
```

The spice class is made up of the string name of the spice, the per unit value, the quantity, and the next spice in the list. In this class, we just need to initialize all of these and create setters and getters. **One thing to note is when creating the getter for perUnitValue() (line 30) we need to make sure we are dividing the unit value by the quantity because the text file will give us the value of the spice for the entire quantity, not just one unit.

7 List of Spices Class

```
1
2   public class ListOfSpices {
3
4           private Spice myHead;
5           private Spice ordered;
6
7           public ListOfSpices()
8           {
9                   myHead = null;
10                  ordered = null;
11          }//List of Spices
12
13          public void add(String name, double value, int quant)
14          {
15                  Spice newSpice = new Spice(name, value, quant);
16                  Spice newNext = myHead;
17                  myHead = newSpice;
18                  myHead.setNext(newNext);
19          }
20
21          public Spice getHead()
22          {
23                  if(isEmpty() == false)
24                  {
25                          return myHead;
26                  }
27                  else
28                          return null;
29          }//getHead
30
31          public boolean isEmpty()
32          {
33                  if (myHead ==null)
34                          return true;
35                  else
36                          return false;
37          }//isEmpty
```

```
38
39          public void spiceSort()
40          {
41                  ordered = null;
42                  Spice curr = myHead;
43                  while(curr != null)
44                  {
45                          Spice next = curr.getNext();
46                          insert(curr);
47                          curr = next;
48                  }//while
49          }//spiceSort
50
51          public void insert(Spice newSpice)
52          {
53                  if(ordered == null || ordered.getPerUnitValue()<=
54                  newSpice.getPerUnitValue())
55                  {
56                          newSpice.setNext(ordered);
57                          ordered = newSpice;
58                  }//if
59                  else
60                  {
61                          Spice curr = ordered;
62                          while(curr.getNext() != null &&
63                          curr.getNext().getPerUnitValue()>newSpice.getPerUnitValue())
64                          {
65                                  curr = curr.getNext();
66                          }//while
67                          newSpice.setNext(curr.getNext());
68                          curr.setNext(newSpice);
69                  }//else
70          }//insert
71
72  }//List of Spices
```

The List of Spices class will allow us to go through the spices in the order of value. We first add all of the spices linking them with myNext. We create a getter for myHead (line 21) to get the first spice in the list, an isEmpty() function (line 31) that tells us if the list is empty, and a spiceSort() method (line 39). Spice sort is called once all of the spices have been added. SpiceSort and insert allow us to order the spices by perUnitValue so the most valuable spices are first and we can claim those first. Ordering the spices makes our greedy algorithm much more simple.

8 Knapsack Class

```
1
2   public class Knapsack {
3
4           private int capacity;
5           private double currentQuant;
6           private double currentVal;
7
8           public Knapsack()
9           {
```

```
                capacity = 0;
                currentQuant = 0;
                currentVal = 0;
        }//knapsack

        public Knapsack(int newCapacity)
        {
                capacity = newCapacity;
                currentQuant = 0;
                currentVal = 0;
        }//Knapsack

        public void setCurrentQuant(double newQuant)
        {
                currentQuant = currentQuant + newQuant;
        }//setCurrentQuant

        public void setCurrentVal(double newVal)
        {
                currentVal = currentVal + newVal;
        }//setCurrentVal

        public int getCapacity()
        {
                return capacity;
        }//getCapacity

        public double getCurrentQuant()
        {
                return currentQuant;
        }//getCurrentQuant

        public double getCurrentVal()
        {
                return currentVal;
        }//getCurrentVal
}//Knapsack
```

The Knapsack class is used to represent the knapsacks we have to hold the spices. Knapsacks are made up of a capacity, a current quantity/current capacity, and a current value (in quatloos of course). This class is also pretty simple: initialize, setters and getters.

9 ...In Main Method

```
public class main {

        public static void main(String[] args)
        {
        try
        {
                System.out.println("Spice Heist: ");
                String file2Name = "spice.txt";
                Scanner readFile = new Scanner (new File (file2Name));
                String text = "";
                String spice [];
                String knapsack [];
```

```
13                  int sackNum = 0;
14                  double difference;
15                  double amount;
16                  double quatloos = 0;
17                  int counter = 0;
18                  String contains = "";
19                  Knapsack[] sacks = new Knapsack[5];
20                  ListOfSpices newSpice = new ListOfSpices();
21                  while(readFile.hasNextLine())
22                  {
23                          text = readFile.nextLine();
24                          if( text.equals("") ||text.charAt(0)!= '-' )
25                          {
26                                  //if line contains spice we are adding a new spice
27                                  if(text.contains("spice"))
28                                  {
29                                          text = text.replace(';', ' ');
30                                          spice = text.split("\\s+");
31                                          newSpice.add(spice[3],
32                                          Double.parseDouble(spice[6]),
33                                          Integer.parseInt(spice[9]));
34
35                                  }//if
36                                  else if(text.contains("knapsack"))
37                                  {
38                                          //if line contains knapsack  we want to know
39                                          //the highest possible
40                                          //value for that knapsack
41                                          text = text.replace(';', ' ');
42                                          knapsack = text.split("\\s+");
43                                          sacks[sackNum]= new
44                                          Knapsack(Integer.parseInt(knapsack[3]));
45                                          sackNum++;
46                                  }//elseIf
47                          }//if
48
49                  }//while
50
51                  //put spices in order by value
52                  newSpice.spiceSort();
53                  Spice curr = newSpice.getHead();
54
55                  while(counter < sacks.length && sacks[counter] != null)
56                  {
57                          //if spice is not null and sack is not full
58                          if(curr != null && sacks[counter].getCurrentQuant() !=
59                          sacks[counter].getCapacity())
60                          {
61                                  //if we can add the entirety of the spice
62                                  if (sacks[counter].getCurrentQuant() +
63                                  curr.getQuant() < sacks[counter].getCapacity())
64                                  {
65                                          sacks[counter].setCurrentQuant(
66                                          curr.getQuant());
67                                          contains = contains + (", " +
68                                          (curr.getQuant()* 1.0) + " scoop(s) of " +
69                                          curr.getName());
70                                          quatloos = quatloos + (curr.getQuant() *
```

```
                                        curr.getPerUnitValue());
                                        curr = curr.getNext();
                                }//if
                                //if we can only add some of the spice
                                else
                                {
                                        difference = (sacks[counter].getCapacity()
                                        - sacks[counter].getCurrentQuant());
                                        amount = difference / curr.getQuant();
                                        sacks[counter].setCurrentQuant(amount *
                                        curr.getQuant());
                                        sacks[counter].setCurrentVal(amount *
                                        curr.getPerUnitValue());
                                        contains = contains +  (", " + amount *
                                        curr.getQuant() + " scoop(s) of "
                                        + curr.getName() );
                                        quatloos = quatloos + ((amount *
                                        curr.getQuant())*
                                        curr.getPerUnitValue());
                                        curr = curr.getNext();
                                }//else
                        }//if

                        else
                        {
                                System.out.println("Knapsack of capacity "
                                + sacks[counter].getCapacity()
                                + " is worth "
                                + quatloos + " quatloos and contains"
                                + contains + ". ");
                                counter ++;
                                curr = newSpice.getHead();
                                quatloos = 0;
                                contains = "";
                        }
                }//while

        }//try
        catch(FileNotFoundException ex)
        {
        System.out.println("Failed to find file: "+ fileName);
        }//catch
        catch(InputMismatchException ex)
                {
                System.out.println("Type mismatch. ");
                System.out.println(ex.getMessage());
                }//catch
        catch(NumberFormatException ex)
                {
                System.out.println("Failed to convert String into an integer. ");
                System.out.println(ex.getMessage());
                }//catch
        catch(NullPointerException ex)
                {
                System.out.println("Null pointer exception. ");
                System.out.println(ex.getMessage());
                }//catch
        catch(Exception ex)
```

```
129                     {
130                     System.out.println("Oops, something went wrong. ");
131                     ex.printStackTrace();
132                     }//catch
133 
134             }//main
135 
136 }//main
```

In main again we start by reading the file. Ignoring the comments, if the line contains spice we first replace all semicolons with spaces to eliminate them from the parse. Then we split on space and create a new spice with the spice's name, price and quantity. If the line contains knapsack we are creating a new knapsack and adding it to an array of knapsacks to traverse later. We also have a counter for sackNum to indicate where in the array to insert the sack, so this must be incremented too. Next we order the spices with spiceSort and set current to be the head of the spice list. While we still have knapsacks to fill we want to first check that the spice is not null and that the sack is not full. Then we want to check if we can add the entirety of the next spice or only a portion. If the sack's current quantity plus the spice's current quantity is less than the sack's capacity we can add all of the current spice. We then adjust the current quantity and value of the sack. We also have a string contains here that keeps track of what is in the sack, so we update that here too. If all of the current spice will not fit in the sack we move on to the else statement and add the portion of the spice that will fit. We find out how much room is left in the sack and divide that by the amount of spice that we have. We then make the same updates as before. Finally, once the sack is full, we print out the capacity of the sack, how many quatloos it is worth, and what how much of which spices are in it. We then set curr to be the head of the spice list again (the most valuable spice), quatloos back to zero and contains to be the empty string. We also increment counter to show up moving onto the next knapsack.

10 Conclusion

The asymptotic running time of the Bellman-Ford Single Source Shortest Path is O(VE), v indicating number of vertices and e indicating number of edges. This is because for each of the vertices (-1 being the source) we have to check all edges to determine which yields the lowest cost. As for the fractional knapsack, we have a running time of O(n log(n)). This is because we must first sort the spices by price then we traverse through the list until the knapsack is full.