

DAA Assignment 3

Name = Mohit Akhouri

Roll No. = 19ucc023

Date Submitted = 24/04/2021

Contents

Sr. No.	Subject	Page No.
1	Assignment part 1 : AcyclicLP and longest path	3-20
1.1	Modification in the code	3-12
1.2	Random parallel job scheduling table and PERT chart	13-15
1.3	Explanation of execution of code and PERT chart	16-18
1.4	Explanation of code's output	19-20
2	Assignment part 2 : BellmanFordSP and –ve cycle	21-50
2.1	Modification in the code	21-40
2.2	Random currency exchange table and resulting Directed graph	41-42
2.3	Explanation of execution of code and graph	43-45
2.4	Explanation of code's output	46-50
3	Conclusion	51

Assignment part 1 : Reducing code of Acyclic Shortest Path and applying code on PERT chart of Random parallel job scheduling problem to detect longest path.

(a) Demonstrate the modification in the code and highlight it :

Code for AcyclicLP is as follows :

```

/******  

*****

```

- ```
* Compilation: javac AcyclicLP.java
* Execution: java AcyclicLP V E
* Dependencies: EdgeWeightedDigraph.java DirectedEdge.java Topological.java
* Data files: https://algs4.cs.princeton.edu/44sp/tinyEWDAG.txt
*
* Computes Longest paths in an edge-weighted acyclic digraph.
*
* Test case 1 : % java AcyclicLP tinyEWDAG.txt 5
*
* Output is :
* 5 to 0 (2.44) 5->1 0.32 1->3 0.29 3->6 0.52 6->4 0.93 4->0 0.38
* 5 to 1 (0.32) 5->1 0.32
* 5 to 2 (2.77) 5->1 0.32 1->3 0.29 3->6 0.52 6->4 0.93 4->7 0.37 7->2 0.34
* 5 to 3 (0.61) 5->1 0.32 1->3 0.29
* 5 to 4 (2.06) 5->1 0.32 1->3 0.29 3->6 0.52 6->4 0.93
* 5 to 5 (0.00)
* 5 to 6 (1.13) 5->1 0.32 1->3 0.29 3->6 0.52
```

\* 5 to 7 (2.43) 5->1 0.32 1->3 0.29 3->6 0.52 6->4 0.93 4->7 0.37

\*

\* Test case 2 : % java AcyclicLP lp1.txt 0

\* lp1.txt is as follows :

\* 8

\* 13

\* 0 1 3

\* 0 2 6

\* 1 2 4

\* 1 4 11

\* 2 6 11

\* 1 3 4

\* 2 3 8

\* 3 4 -4

\* 3 5 5

\* 3 6 2

\* 4 7 9

\* 5 7 1

\* 6 7 2

\*

\* Output is :

\* 0 to 0 (0.00)

\* 0 to 1 (3.00) 0->1 3.00

\* 0 to 2 (7.00) 0->1 3.00 1->2 4.00

\* 0 to 3 (15.00) 0->1 3.00 1->2 4.00 2->3 8.00

\* 0 to 4 (14.00) 0->1 3.00 1->4 11.00

```
* 0 to 5 (20.00) 0->1 3.00 1->2 4.00 2->3 8.00 3->5 5.00
* 0 to 6 (18.00) 0->1 3.00 1->2 4.00 2->6 11.00
* 0 to 7 (23.00) 0->1 3.00 1->4 11.00 4->7 9.00
```

```

*****/
```

```
// Roll no = 19ucc023
```

```
// Name = Mohit Akhouri
```

```
// DAA Assignment 3 - Acyclic Longest Path and Parallel Job scheduling algorithm
```

```
// Date submitted = 23/4/2021
```

```
//import edu.princeton.cs.algs4.AcyclicLP;
```

```
import edu.princeton.cs.algs4.DirectedEdge;
```

```
import edu.princeton.cs.algs4.EdgeWeightedDigraph;
```

```
import edu.princeton.cs.algs4.In;
```

```
import edu.princeton.cs.algs4.Stack;
```

```
import edu.princeton.cs.algs4.StdOut;
```

```
import edu.princeton.cs.algs4.Topological;
```

```
public class AcyclicLP {
```

```
 private double[] distTo; // distTo[v] = distance of longest s->v path
```

```
 private DirectedEdge[] edgeTo; // edgeTo[v] = last edge on longest s->v path
```

```
 /**
```

```
 * Computes a longest paths tree from {@code s} to every other vertex in
```

```
 * the directed acyclic graph {@code G}.
```

- \* @param G the acyclic digraph
- \* @param s the source vertex
- \* @throws IllegalArgumentException if the digraph is not acyclic
- \* @throws IllegalArgumentException unless {@code 0 ≤ s < V}
- \*/

```
public AcyclicLP(EdgeWeightedDigraph G, int s) {
 distTo = new double[G.V()];
 edgeTo = new DirectedEdge[G.V()];

 validateVertex(s);

 for (int v = 0; v < G.V(); v++)
 distTo[v] = Double.NEGATIVE_INFINITY;
 distTo[s] = 0.0;

 // visit vertices in topological order
 Topological topological = new Topological(G);
 if (!topological.hasOrder())
 throw new IllegalArgumentException("Digraph is not acyclic.");
 for (int v : topological.order()) {
 for (DirectedEdge e : G.adj(v))
 relax(e);
 }
}

// relax edge e
```

```

private void relax(DirectedEdge e) {
 int v = e.from(), w = e.to();
 if (distTo[w] < distTo[v] + e.weight()) {
 distTo[w] = distTo[v] + e.weight();
 edgeTo[w] = e;
 }
}

```

```

/**

```

\* Returns the length of a longest path from the source vertex {@code s} to vertex {@code v}.

\* @param v the destination vertex

\* @return the length of a longest path from the source vertex {@code s} to vertex {@code v};

\* {@code Double.NEGATIVE\_INFINITY} if no such path

\* @throws IllegalArgumentException unless {@code 0 <= v < V}

```

*/

```

```

public double distTo(int v) {
 validateVertex(v);
 return distTo[v];
}

```

```

/**

```

\* Is there a path from the source vertex {@code s} to vertex {@code v}?

\* @param v the destination vertex

\* @return {@code true} if there is a path from the source vertex

\* {@code s} to vertex {@code v}, and {@code false} otherwise

```

* @throws IllegalArgumentException unless {@code 0 <= v < V}
*/
public boolean hasPathTo(int v) {
 validateVertex(v);
 return distTo[v] > Double.NEGATIVE_INFINITY;
}

/**
 * Returns a longest path from the source vertex {@code s} to vertex {@code
v}.
 * @param v the destination vertex
 * @return a longest path from the source vertex {@code s} to vertex {@code
v}
 * as an iterable of edges, and {@code null} if no such path
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public Iterable<DirectedEdge> pathTo(int v) {
 validateVertex(v);
 if (!hasPathTo(v)) return null;
 Stack<DirectedEdge> path = new Stack<DirectedEdge>();
 for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
 path.push(e);
 }
 return path;
}

```

```
// throw an IllegalArgumentException unless {@code 0 <= v < V}
```



```

private void validateVertex(int v) {
 int V = distTo.length;
 if (v < 0 || v >= V)
 throw new IllegalArgumentException("vertex " + v + " is not between 0
and " + (V-1));
}

```

```

public static void main(String[] args) {
 In in = new In(args[0]);
 int s = Integer.parseInt(args[1]);
 EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);

```

```

// find longest path from s to each other vertex in DAG

```

```

AcyclicLP lp = new AcyclicLP(G, s);

```

```

for (int v = 0; v < G.V(); v++) {

```

```

 if (lp.hasPathTo(v)) {

```

```

 StdOut.printf("%d to %d (%.2f) ", s, v, lp.distTo(v));

```

```

 for (DirectedEdge e : lp.pathTo(v)) {

```

```

 StdOut.print(e + " ");

```

```

 }

```

```

 StdOut.println();

```

```

 } else {

```

```

 StdOut.printf("%d to %d no path\n", s, v);

```

```

 }

```

```

}

```

```

}

```

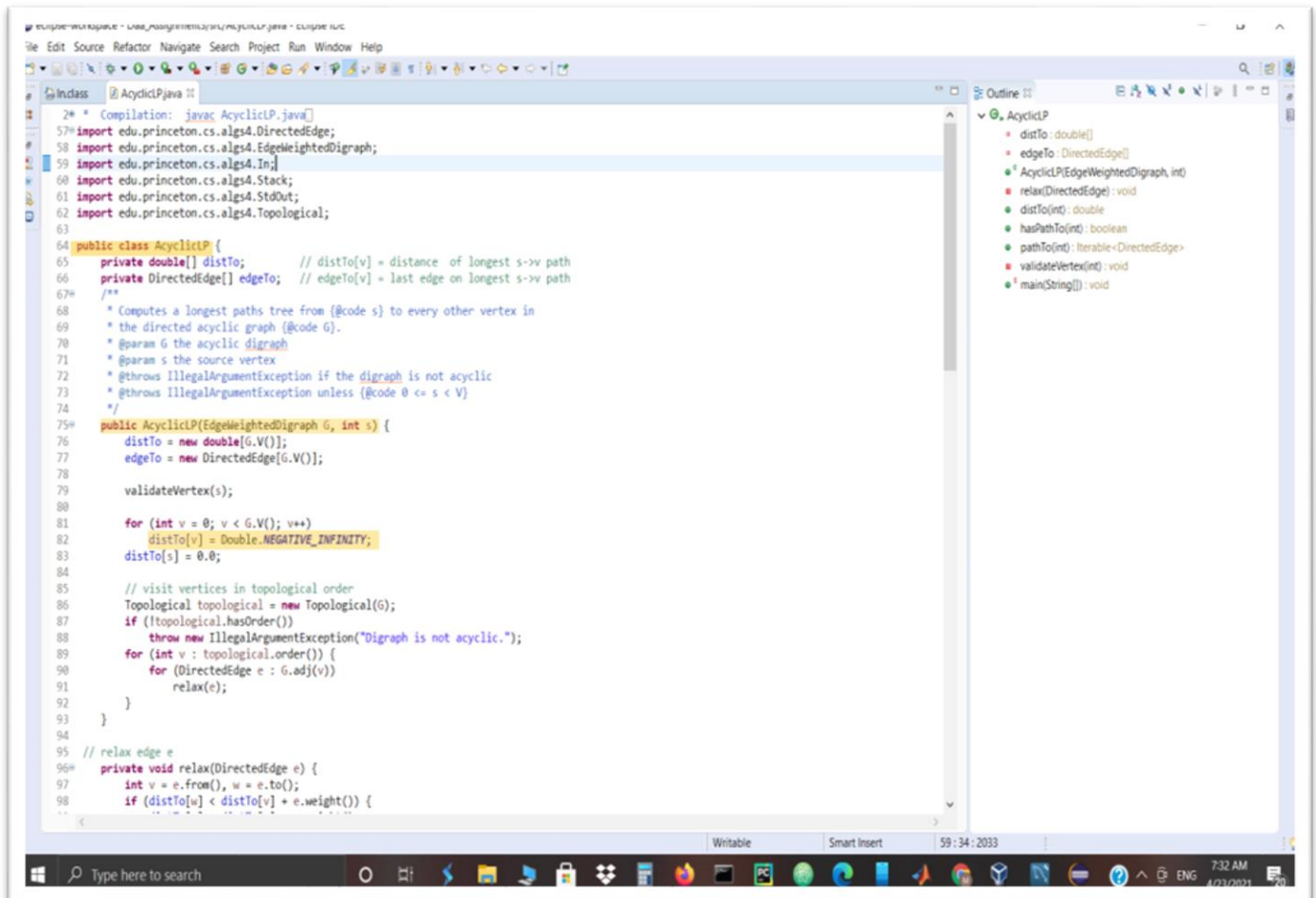
```

}

```

Modifications in the code of AcyclicSP is as follows (Highlighted Lines changed) :

New class designed (AcyclicLP) with same code as AcyclicSP , but with some modifications



```
2 * Compilation: javac AcyclicLP.java
57 import edu.princeton.cs.algs4.DirectedEdge;
58 import edu.princeton.cs.algs4.EdgeWeightedDigraph;
59 import edu.princeton.cs.algs4.In;
60 import edu.princeton.cs.algs4.Stack;
61 import edu.princeton.cs.algs4.StdOut;
62 import edu.princeton.cs.algs4.Topological;
63
64 public class AcyclicLP {
65 private double[] distTo; // distTo[v] = distance of longest s->v path
66 private DirectedEdge[] edgeTo; // edgeTo[v] = last edge on longest s->v path
67
68 /**
69 * Computes a longest paths tree from {@code s} to every other vertex in
70 * the directed acyclic graph {@code G}.
71 * @param G the acyclic digraph
72 * @param s the source vertex
73 * @throws IllegalArgumentException if the digraph is not acyclic
74 * @throws IllegalArgumentException unless {@code 0 <= s < V}
75 */
76 public AcyclicLP(EdgeWeightedDigraph G, int s) {
77 distTo = new double[G.V()];
78 edgeTo = new DirectedEdge[G.V()];
79
80 validateVertex(s);
81
82 for (int v = 0; v < G.V(); v++)
83 distTo[v] = Double.NEGATIVE_INFINITY;
84 distTo[s] = 0.0;
85
86 // visit vertices in topological order
87 Topological topological = new Topological(G);
88 if (!topological.hasOrder())
89 throw new IllegalArgumentException("Digraph is not acyclic.");
90 for (int v : topological.order()) {
91 for (DirectedEdge e : G.adj(v))
92 relax(e);
93 }
94 }
95
96 // relax edge e
97 private void relax(DirectedEdge e) {
98 int v = e.from(), w = e.to();
99 if (distTo[w] < distTo[v] + e.weight()) {
100 distTo[w] = distTo[v] + e.weight();
101 edgeTo[w] = e;
102 }
103 }
104
105 /**
106 * Returns the longest path from s to v.
107 * @param v the vertex
108 * @return the longest path from s to v
109 * @throws IllegalArgumentException unless {@code 0 <= v < V}
110 */
111 public Iterable<DirectedEdge> pathTo(int v) {
112 if (!hasPathTo(v))
113 throw new IllegalArgumentException("No path from s to " + v);
114 return pathTo(v);
115 }
116
117 /**
118 * Returns the longest path from s to v.
119 * @param v the vertex
120 * @return the longest path from s to v
121 * @throws IllegalArgumentException unless {@code 0 <= v < V}
122 */
123 public double distTo(int v) {
124 if (!hasPathTo(v))
125 throw new IllegalArgumentException("No path from s to " + v);
126 return distTo[v];
127 }
128
129 /**
130 * Returns true if there is a path from s to v.
131 * @param v the vertex
132 * @return true if there is a path from s to v
133 * @throws IllegalArgumentException unless {@code 0 <= v < V}
134 */
135 public boolean hasPathTo(int v) {
136 return distTo[v] > Double.NEGATIVE_INFINITY;
137 }
138
139 /**
140 * Returns the longest path from s to v.
141 * @param v the vertex
142 * @return the longest path from s to v
143 * @throws IllegalArgumentException unless {@code 0 <= v < V}
144 */
145 public DirectedEdge edgeTo(int v) {
146 if (!hasPathTo(v))
147 throw new IllegalArgumentException("No path from s to " + v);
148 return edgeTo[v];
149 }
150
151 /**
152 * Returns the longest path from s to v.
153 * @param v the vertex
154 * @return the longest path from s to v
155 * @throws IllegalArgumentException unless {@code 0 <= v < V}
156 */
157 public void validateVertex(int v) {
158 if (v < 0 || v >= G.V())
159 throw new IllegalArgumentException("vertex " + v + " is not a valid index of " + G);
160 }
161
162 /**
163 * Returns the longest path from s to v.
164 * @param v the vertex
165 * @return the longest path from s to v
166 * @throws IllegalArgumentException unless {@code 0 <= v < V}
167 */
168 public void main(String[] args) {
169 In in = new In(args[0]);
170 EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
171 AcyclicLP lp = new AcyclicLP(G, Integer.parseInt(args[1]));
172 for (int v = 0; v < G.V(); v++)
173 if (!lp.hasPathTo(v))
174 StdOut.println("No path from s to " + v);
175 else
176 StdOut.println("Longest path from s to " + v + " is " + lp.pathTo(v));
177 }
178 }
```

Lines changed : Line 64,75 and 82

What is changed : Class name changed from AcyclicSP to AcyclicLP , constructor name also change from AcyclicSP to AcyclicLP and constant **POSITIVE\_INFINITY** changed to **NEGATIVE\_INFINITY**.

Reason to change : For the code of AcyclicSP to detect longest path, constant **NEGATIVE\_INFINITY** is to be used since edge weights are negated and so their distance becomes **NEGATIVE\_INFINITY** from the source vertex.

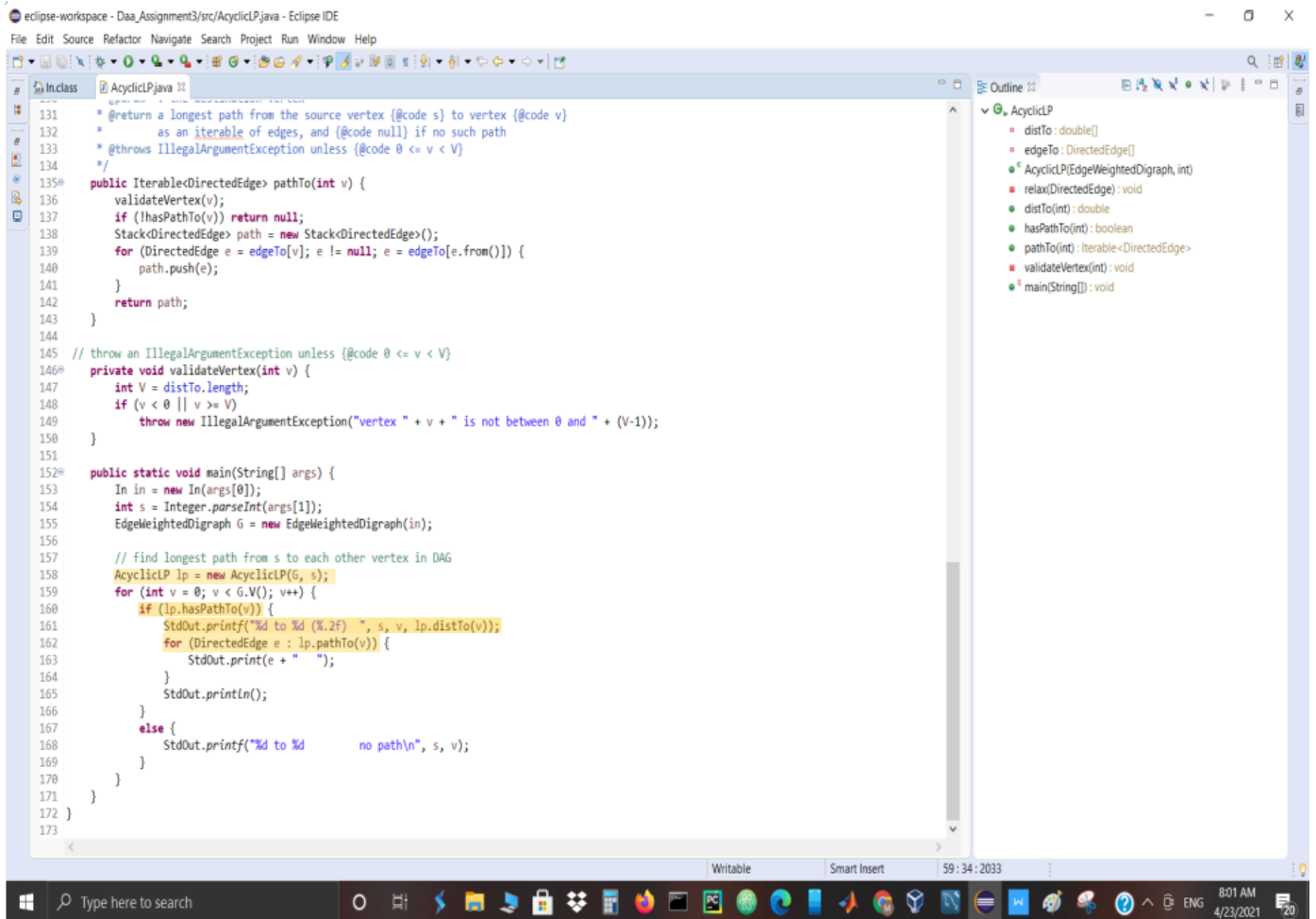
```
95 // relax edge e
96 private void relax(DirectedEdge e) {
97 int v = e.from(), w = e.to();
98 if (distTo[w] < distTo[v] + e.weight()) {
99 distTo[w] = distTo[v] + e.weight();
100 edgeTo[w] = e;
101 }
102 }
103
104 /**
105 * Returns the length of a longest path from the source vertex {@code s} to vertex {@code v}.
106 * @param v the destination vertex
107 * @return the length of a longest path from the source vertex {@code s} to vertex {@code v};
108 * {@code Double.NEGATIVE_INFINITY} if no such path
109 * @throws IllegalArgumentException unless {@code 0 <= v < V}
110 */
111 public double distTo(int v) {
112 validateVertex(v);
113 return distTo[v];
114 }
115
116 /**
117 * Is there a path from the source vertex {@code s} to vertex {@code v}?
118 * @param v the destination vertex
119 * @return {@code true} if there is a path from the source vertex
120 * {@code s} to vertex {@code v}, and {@code false} otherwise
121 * @throws IllegalArgumentException unless {@code 0 <= v < V}
122 */
123 public boolean hasPathTo(int v) {
124 validateVertex(v);
125 return distTo[v] > Double.NEGATIVE_INFINITY;
126 }
127
128 /**
129 * Returns a longest path from the source vertex {@code s} to vertex {@code v}.
130 * @param v the destination vertex
131 * @return a longest path from the source vertex {@code s} to vertex {@code v}
132 * as an iterable of edges, and {@code null} if no such path
133 * @throws IllegalArgumentException unless {@code 0 <= v < V}
134 */
135 public Iterable<DirectedEdge> pathTo(int v) {
136 validateVertex(v);
137 if (!hasPathTo(v)) return null;
138 }
```

Lines changed : Line 98 and 125

What is changed : The Relaxation condition is changed from  $v.d > u.d + W(u,v)$  to  $v.d < u.d + W(u,v)$ .

Where  $(u,v)$  is a edge and  $u,v$  are vertices,  $v.d$  is the distance of final vertex 'v' and  $u.d$  is the distance of initial vertex 'u' and  $W(u,v)$  is the weight of edge  $(u,v)$ . and constant changed to **NEGATIVE\_INFINITY** and '<' symbol becomes '>'.

Reason to change : According to the theory taught in DAA class, for longest path, just reverse the relaxation condition and check if  $v.d < u.d + W(u,v)$  then replace  $v.d$  with **longest distance** i.e,  $u.d + W(u,v)$  . Now since constant changed from **POSITIVE\_INFINITY** to **NEGATIVE\_INFINITY** , change the '<' symbol to '>' symbol so that it can detect the path correctly.



```
131 * @return a longest path from the source vertex {@code s} to vertex {@code v}
132 * as an iterable of edges, and {@code null} if no such path
133 * @throws IllegalArgumentException unless {@code 0 <= v < V}
134 */
135 public Iterable<DirectedEdge> pathTo(int v) {
136 validateVertex(v);
137 if (!hasPathTo(v)) return null;
138 Stack<DirectedEdge> path = new Stack<DirectedEdge>();
139 for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
140 path.push(e);
141 }
142 return path;
143 }
144
145 // throw an IllegalArgumentException unless {@code 0 <= v < V}
146 private void validateVertex(int v) {
147 int V = distTo.length;
148 if (v < 0 || v >= V)
149 throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
150 }
151
152 public static void main(String[] args) {
153 In in = new In(args[0]);
154 int s = Integer.parseInt(args[1]);
155 EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
156
157 // find longest path from s to each other vertex in DAG
158 AcyclicLP lp = new AcyclicLP(G, s);
159 for (int v = 0; v < G.V(); v++) {
160 if (lp.hasPathTo(v)) {
161 StdOut.printf("%d to %d (%.2f) ", s, v, lp.distTo(v));
162 for (DirectedEdge e : lp.pathTo(v)) {
163 StdOut.print(e + " ");
164 }
165 StdOut.println();
166 }
167 else {
168 StdOut.printf("%d to %d no path\n", s, v);
169 }
170 }
171 }
172 }
173 }
```

Outline:

- AcyclicLP
  - distTo: double[]
  - edgeTo: DirectedEdge[]
  - AcyclicLP(EdgeWeightedDigraph, int)
  - relax(DirectedEdge): void
  - distTo(int): double
  - hasPathTo(int): boolean
  - pathTo(int): Iterable<DirectedEdge>
  - validateVertex(int): void
  - main(String[]): void

Lines changed : Lines 158, 160, 161 and 162.

What is changed : Object names changed from **sp** to **lp** . Object declaration is in terms of new class **AcyclicLP**.

Reason to change : just for convention

**( b ) Demonstrate the random job-scheduling problem table and its resulting PERT chart :**

*(I) - The random parallel job-scheduling table is as follows :*

| JOB | DURATION | MUST COMPLETE BEFORE |
|-----|----------|----------------------|
| 0   | 9        | 1,2                  |
| 1   | 11       | 3,4                  |
| 2   | 10       | 4                    |
| 3   | 12       | 5                    |
| 4   | 13       | 5                    |
| 5   | 10       | -                    |

**Explanation of the above parallel-job scheduling table :**

The Parallel Job scheduling table consists of 6 jobs = job 1 to job 5

Duration of each job is given in the **DURATION** column

Must complete before means :

Job 0 should be done before jobs 1 and 2

Job 1 should be done before jobs 3 and 4

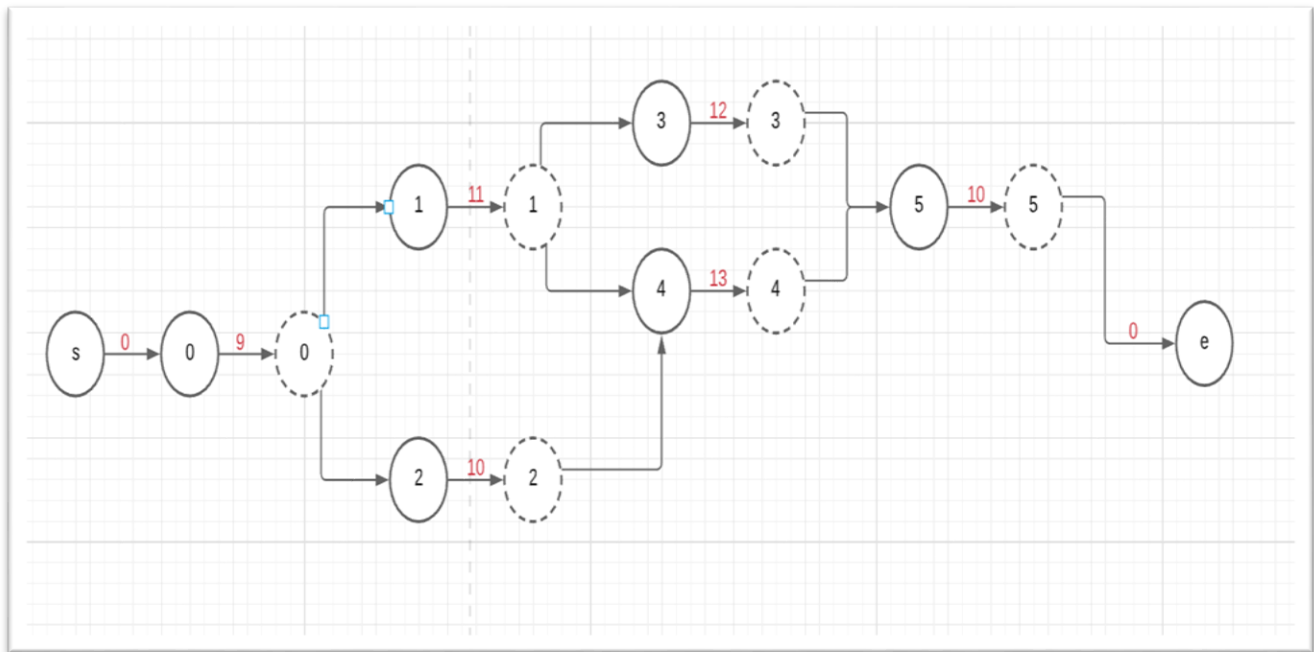
Job 2 should be done before job 4

Job 3 should be done before job 5

Job 4 should be done before job 5

Job 5 has no successor, it can be done before any of the jobs

(II) - The resulting PERT chart of the above table is as follows :



Explanation of the above parallel-job scheduling PERT chart :

The PERT chart consists of 8 main ( **solid circles** ) vertices and 6 subsidiary ( **dashed circles** ) .

Each node has **three edges** : **starting edge**, **duration edge** and **destination edge**.

For ex:

Node 1 has starting edge from dashed 0 vertex to solid 1 vertex

Node 1 has duration edge from solid 1 vertex to dashed 1 vertex of **weight = 11**

Node 1 has destination edge from dashed 1 vertex to solid 3 and 4

Now if a vertex ( say vertex 1 ) is drawn before another vertices/vertex ( say vertex 3 and 4 ), then it means **“Job 1 should complete before Jobs 3 and 4”**.

What is CRITICAL PATH in the above PERT chart :

Critical path means calculating the **longest path** ( i.e, maximum possible time required to complete all jobs ) .

So in the above PERT chart , Critical path is as follows :

s -> 0 -> 1 -> 4 -> 5 -> e

Total time calculation :

s -> 0 = 0 units

0 -> 1 = 9 units

1 -> 4 = 11 units

4 -> 5 = 13 units

5 -> e = 10 units

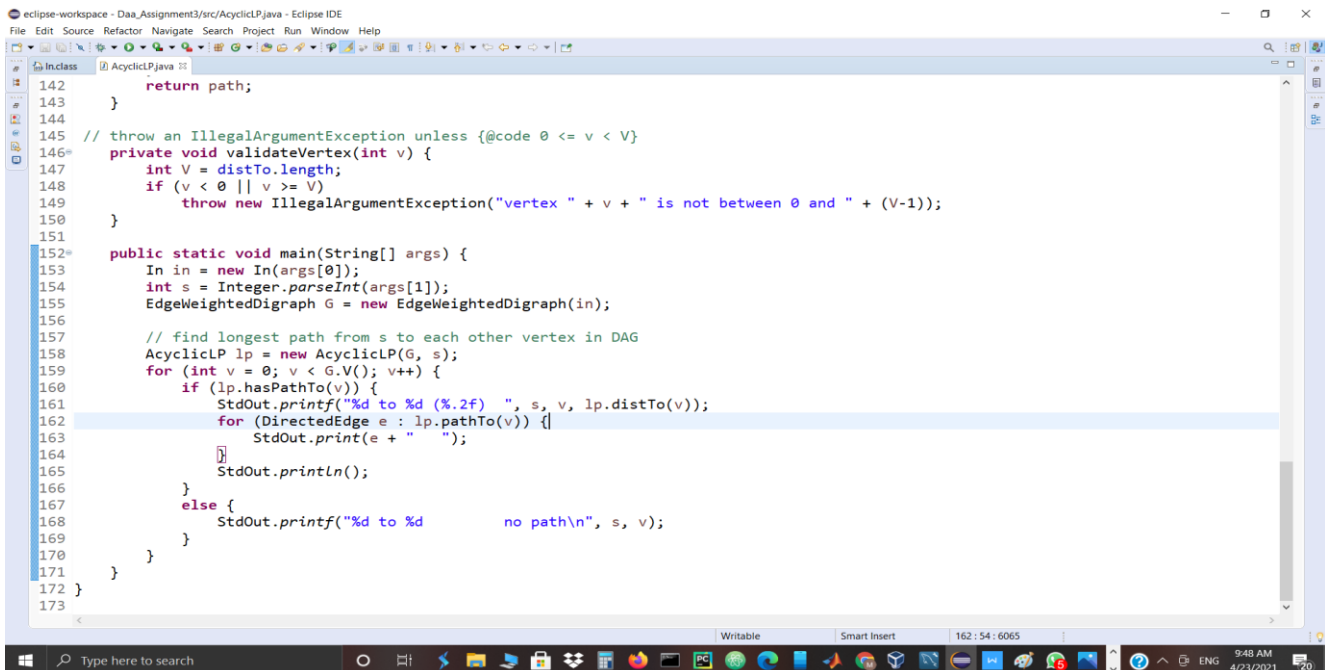
**Total time = 0+9+11+13+10 = 43 units**

Hence , 43 units of time is required in completing all the jobs as derived from the PERT chart .

Now this can be implemented by means of a **Directed Acyclic graph** by considering **jobs as vertices** and **duration as edges** and applying the AcyclicLP code on the graph .

AcyclicLP will compute the Longest path in the graph, thus giving the maximum possible time to complete all the jobs . This will be demonstrated by means of code in section 3 below .

( c ) Demonstrate the execution of the code and on the above PERT chart :



```
142 return path;
143 }
144
145 // throw an IllegalArgumentException unless (0 <= v < V)
146 private void validateVertex(int v) {
147 int V = distTo.length;
148 if (v < 0 || v >= V)
149 throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
150 }
151
152 public static void main(String[] args) {
153 In in = new In(args[0]);
154 int s = Integer.parseInt(args[1]);
155 EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
156
157 // find longest path from s to each other vertex in DAG
158 AcyclicLP lp = new AcyclicLP(G, s);
159 for (int v = 0; v < G.V(); v++) {
160 if (lp.hasPathTo(v)) {
161 StdOut.printf("%d to %d (%.2f) ", s, v, lp.distTo(v));
162 for (DirectedEdge e : lp.pathTo(v)) {
163 StdOut.print(e + " ");
164 }
165 StdOut.println();
166 }
167 else {
168 StdOut.printf("%d to %d no path\n", s, v);
169 }
170 }
171 }
172 }
173 }
```

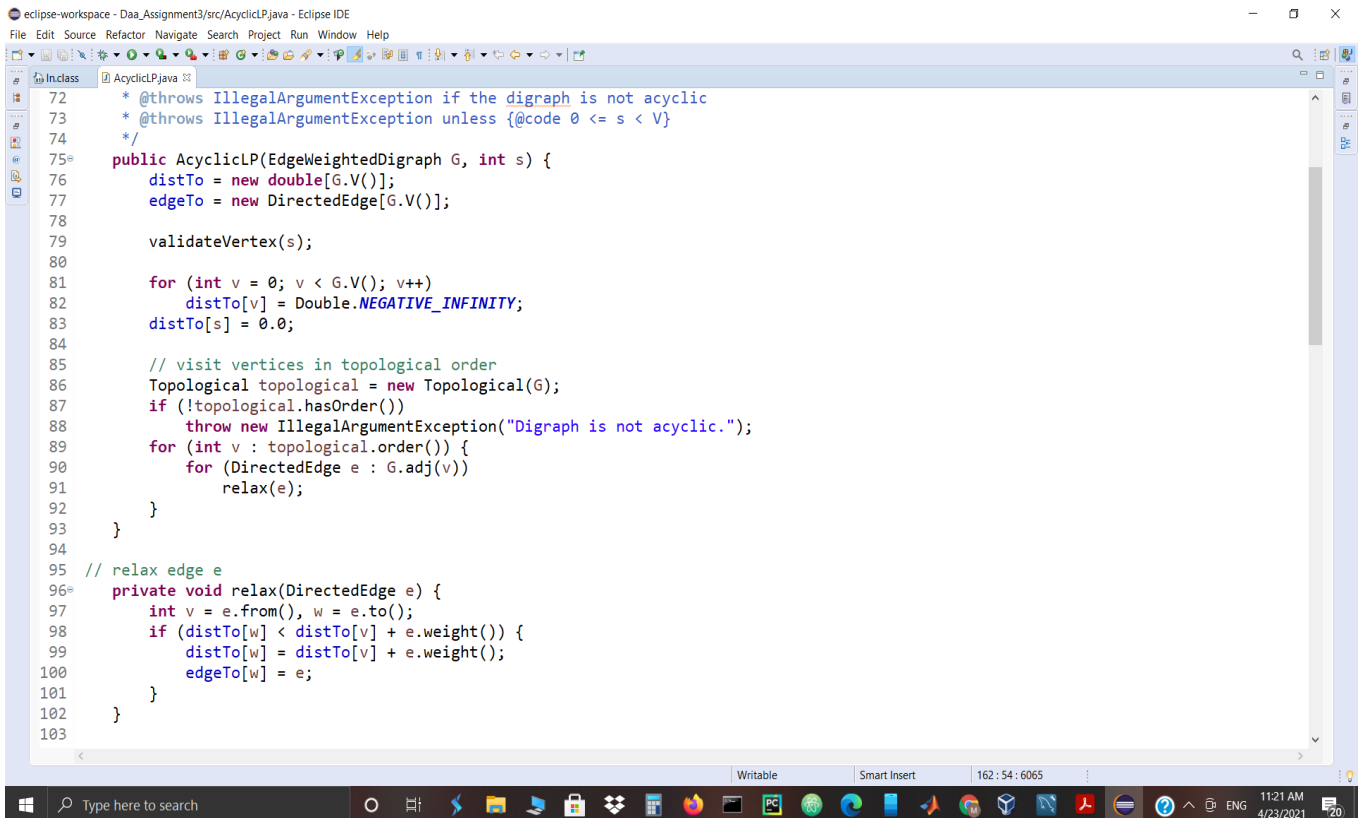
Code execution starts from the main function. Input is given a **textfile** that contains the following information :

- Number of vertices
- Number of edges
- Edges and their weights

Text-file is given as args[0] and starting vertex from where longest path is to be calculated as args[1]. Now a edge-weighted graph is created with the help of input text file. Now object of AcyclicLP is created to find longest path.

In the printing part, the longest path is calculated between the source and different vertices, and if no path exists, then simple message of “no path” is printed for those set of vertices.





```
72 * @throws IllegalArgumentException if the digraph is not acyclic
73 * @throws IllegalArgumentException unless {code 0 <= s < V}
74 */
75 public AcyclicLP(EdgeWeightedDigraph G, int s) {
76 distTo = new double[G.V()];
77 edgeTo = new DirectedEdge[G.V()];
78
79 validateVertex(s);
80
81 for (int v = 0; v < G.V(); v++)
82 distTo[v] = Double.NEGATIVE_INFINITY;
83 distTo[s] = 0.0;
84
85 // visit vertices in topological order
86 Topological topological = new Topological(G);
87 if (!topological.hasOrder())
88 throw new IllegalArgumentException("Digraph is not acyclic.");
89 for (int v : topological.order()) {
90 for (DirectedEdge e : G.adj(v))
91 relax(e);
92 }
93 }
94
95 // relax edge e
96 private void relax(DirectedEdge e) {
97 int v = e.from(), w = e.to();
98 if (distTo[w] < distTo[v] + e.weight()) {
99 distTo[w] = distTo[v] + e.weight();
100 edgeTo[w] = e;
101 }
102 }
103
```

Now the functions in the above image are as follows :

- **Relax(DirectedEdge e)** : It applies the relaxation property on the edge . So that in every iteration , the longest path value is substituted in place of old value of vertex. The relaxation property is as follows :

If (  $v.d < u.d + W(u,v)$  ) then

$v.d = u.d + W(u,v)$

$v.\pi = u$

Now the above equations means , if  $u$  and  $v$  are start and end vertices of edge, then if total of **start and weight of edge** is more than **end vertex distance** then, replace  $v.d$  by sum of  $u.d$  and  $W(u,v)$  . Also make ' $u$ ' as the new parent of ' $v$ '. This is called as relaxation property.

- AcyclicLP ( EdgeWeightedDigraph G, int s ) : Here s is the starting vertex from where the longest path to all other vertices are to be calculated. Now initially The distance of source vertex 's' is taken to be zero and distance of all other vertices from 's' to be  $\infty$  . Now all the vertices are sorted in **topological order** and a loop is called on the topologically sorted vertices and relaxation is done for every edge by calling the relax(edge e) function.

The above PERT chart is modelled as a DAG ( directed acyclic graph ) and code is executed as follows with the distances between the matrices calculated and stored in the array as the two arrays below :

where 0 is the source vertex ( 's' from the above PERT chart ) and 7 is the end vertex ( 'e' from the above PERT chart ) .

Initial condition of all vertices, starting vertex = 0 and all other vertices =  $\infty$

|   |          |          |          |          |          |          |          |
|---|----------|----------|----------|----------|----------|----------|----------|
| 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 0 | 1        | 2        | 3        | 4        | 5        | 6        | 7        |

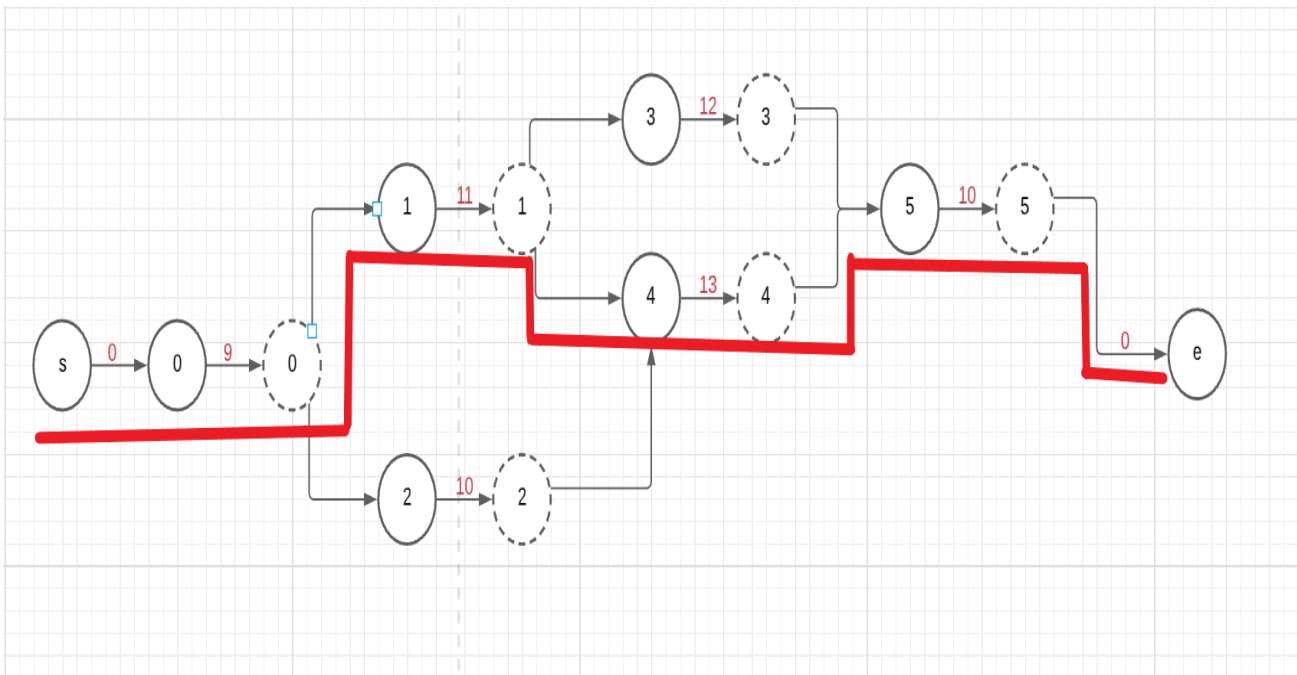
After successfully running the relax function on all edges of vertices arranged in topological order , the final array of vertices and distances is as follows :

|   |   |   |   |    |    |    |    |
|---|---|---|---|----|----|----|----|
| 0 | 0 | 9 | 9 | 20 | 20 | 33 | 43 |
| 0 | 1 | 2 | 3 | 4  | 5  | 6  | 7  |

and hence 43 is the final ( longest path ) from start vertex ( 's' or 0 ) to end vertex ( 'e' or 7 ) .

( d ) Write and describe the longest path in the PERT chart as well as in the code's output result :

*The longest path in the PERT chart is marked with red line in the figure given below :*



Hence the total time is as follows :

Sum of the weights on the **red-line** =  $0 + 9 + 11 + 13 + 10 = 43$

Hence **43** is the **longest path** from source vertex 's' to end vertex 'e' .

*Now the code 'AcyclicLP' is executed on the above PERT chart by modelling the chart as a **Directed Acyclic Graph** with the following specifications :*

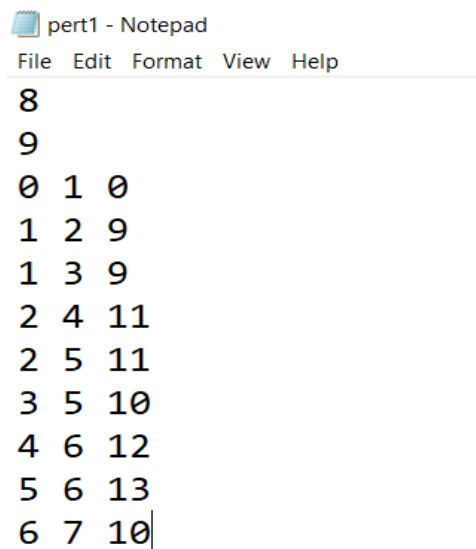
AcyclicLP.java is called on the textfile which is described below and command which was run is :

```
java AcyclicLP pert1.txt 0
```

Where 0 is the starting vertex ( or 's' from the above PERT chart ) .

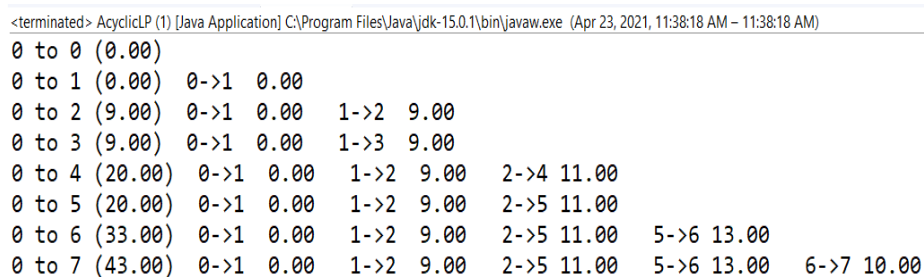
The text file used is pert1.txt which contains :

- Number of vertices : 8
- Number of edges : 9
- Edges and their weights



```
8
9
0 1 0
1 2 9
1 3 9
2 4 11
2 5 11
3 5 10
4 6 12
5 6 13
6 7 10
```

The output of the code is as follows :



```
<terminated> AcyclicLP (1) [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe (Apr 23, 2021, 11:38:18 AM - 11:38:18 AM)
0 to 0 (0.00)
0 to 1 (0.00) 0->1 0.00
0 to 2 (9.00) 0->1 0.00 1->2 9.00
0 to 3 (9.00) 0->1 0.00 1->3 9.00
0 to 4 (20.00) 0->1 0.00 1->2 9.00 2->4 11.00
0 to 5 (20.00) 0->1 0.00 1->2 9.00 2->5 11.00
0 to 6 (33.00) 0->1 0.00 1->2 9.00 2->5 11.00 5->6 13.00
0 to 7 (43.00) 0->1 0.00 1->2 9.00 2->5 11.00 5->6 13.00 6->7 10.00
```

In the above output, 0 is the source vertex (or 's' as in the PERT chart) and 7 is the end vertex (or 'e' as in the PERT chart ) and 43.00 is the longest distance from source vertex ( 0 or 's' ) to end vertex ( 7 or 'e' ). Hence this is the maximum possible time for all jobs to finish.

**Assignment part 2**: Reducing code of Bellman-Ford's algorithm to detect negative cycles for a given graph and applying the reduced code to a random currency exchange problem where negative cycle will depict the capital benefitting due to exchange.

**( a )** Demonstrate the modification in the code and highlight it :

```
import edu.princeton.cs.algs4.DirectedEdge;
```

```
import edu.princeton.cs.algs4.EdgeWeightedDigraph;
```

```
import edu.princeton.cs.algs4.EdgeWeightedDirectedCycle;
```

```
import edu.princeton.cs.algs4.In;
```

```
import edu.princeton.cs.algs4.Queue;
```

```
import edu.princeton.cs.algs4.Stack;
```

```
import edu.princeton.cs.algs4.StdOut;
```

```
import java.io.*;
```

```
import java.util.*;
```

```
/******
```

```

```

```
* Compilation: javac BellmanFordSP.java
```

```
* Execution: java BellmanFordSP filename.txt s
```

```
* Dependencies: EdgeWeightedDigraph.java DirectedEdge.java Queue.java
```

```
* EdgeWeightedDirectedCycle.java
```

```
* Data files: https://algs4.cs.princeton.edu/44sp/tinyEWDn.txt
```

```
* https://algs4.cs.princeton.edu/44sp/mediumEWDnc.txt
```

```
*
```

```

* Bellman-Ford shortest path algorithm. Computes the shortest path tree in
* edge-weighted digraph G from vertex s, or finds a negative cost cycle
* reachable from s.
*
* % java BellmanFordSP tinyEWDn.txt 0
* 0 to 0 (0.00)
* 0 to 1 (0.93) 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52 6->4 -1.25 4->5
0.35 5->1 0.32
* 0 to 2 (0.26) 0->2 0.26
* 0 to 3 (0.99) 0->2 0.26 2->7 0.34 7->3 0.39
* 0 to 4 (0.26) 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52 6->4 -1.25
* 0 to 5 (0.61) 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52 6->4 -1.25 4->5
0.35
* 0 to 6 (1.51) 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52
* 0 to 7 (0.60) 0->2 0.26 2->7 0.34
*
* % java BellmanFordSP tinyEWDnc.txt 0
* 4->5 0.35
* 5->4 -0.66
*
*

*****/

/**
* The {@code BellmanFordSP} class represents a data type for solving the
* single-source shortest paths problem in edge-weighted digraphs with

```

- \* no negative cycles.
- \* The edge weights can be positive, negative, or zero.
- \* This class finds either a shortest path from the source vertex  $s$
- \* to every other vertex or a negative cycle reachable from the source vertex.
- \*
- \* This implementation uses a queue-based implementation of
- \* the Bellman-Ford-Moore algorithm.
- \* The constructor takes  $\Theta(E \cdot V)$  time
- \* in the worst case, where  $V$  is the number of vertices and
- \*  $E$  is the number of edges. In practice, it performs much better.
- \* Each instance method takes  $\Theta(1)$  time.
- \* It uses  $\Theta(V)$  extra space (not including the
- \* edge-weighted digraph).
- \*
- \* This correctly computes shortest paths if all arithmetic performed is
- \* without floating-point rounding error or arithmetic overflow.
- \* This is the case if all edge weights are integers and if none of the
- \* intermediate results exceeds  $2^{52}$ . Since all intermediate
- \* results are sums of edge weights, they are bounded by  $V \cdot C$ ,
- \* where  $V$  is the number of vertices and  $C$  is the
- \* maximum
- \* absolute value of any edge weight.
- \*
- \* For additional documentation,
- \* see [Section 4.4](https://algs4.cs.princeton.edu/44sp) of
- \* *Algorithms, 4th Edition* by Robert Sedgwick and Kevin Wayne.
- \*

\* @author Robert Sedgewick

\* @author Kevin Wayne

\*/

```
public class BellmanFordSP {
```

```
 // for floating-point precision issues
```

```
 private static final double EPSILON = 1E-14;
```

```
 static int size=0;
```

```
 private double[] distTo; // distTo[v] = distance of shortest s->v path
```

```
 private DirectedEdge[] edgeTo; // edgeTo[v] = last edge on shortest s->v
path
```

```
 private boolean[] onQueue; // onQueue[v] = is v currently on the
queue?
```

```
 private Queue<Integer> queue; // queue of vertices to relax
```

```
 private int cost; // number of calls to relax()
```

```
 private Iterable<DirectedEdge> cycle; // negative cycle (or null if no such
cycle)
```

```
/**
```

```
 * Computes a shortest paths tree from {@code s} to every other vertex in
```

```
 * the edge-weighted digraph {@code G}.
```

```
 * @param G the acyclic digraph
```

```
 * @param s the source vertex
```

```
 * @throws IllegalArgumentException unless {@code 0 <= s < V}
```

```
*/
```

```
public BellmanFordSP(EdgeWeightedDigraph G, int s) {
```

```
 distTo = new double[G.V()];
```



```
edgeTo = new DirectedEdge[G.V()];
onQueue = new boolean[G.V()];
for (int v = 0; v < G.V(); v++)
 distTo[v] = Double.POSITIVE_INFINITY;
distTo[s] = 0.0;
```

```
// Bellman-Ford algorithm
```

```
queue = new Queue<Integer>();
queue.enqueue(s);
onQueue[s] = true;
while (!queue.isEmpty() && !hasNegativeCycle()) {
 int v = queue.dequeue();
 onQueue[v] = false;
 relax(G, v);
}
```

```
assert check(G, s);
```

```
}
```

```
// relax vertex v and put other endpoints on queue if changed
```

```
private void relax(EdgeWeightedDigraph G, int v) {
 for (DirectedEdge e : G.adj(v)) {
 int w = e.to();
 if (distTo[w] > distTo[v] + e.weight() + EPSILON) {
 distTo[w] = distTo[v] + e.weight();
 edgeTo[w] = e;
 }
 }
}
```

```

 if (!onQueue[w]) {
 queue.enqueue(w);
 onQueue[w] = true;
 }
 }

 if (++cost % G.V() == 0) {
 findNegativeCycle();
 if (hasNegativeCycle()) return; // found a negative cycle
 }
}
}

```

```
/**
```

```
* Is there a negative cycle reachable from the source vertex {@code s}?
```

```
* @return {@code true} if there is a negative cycle reachable from the
```

```
* source vertex {@code s}, and {@code false} otherwise
```

```
*/
```

```
public boolean hasNegativeCycle() {
```

```
 return cycle != null;
```

```
}
```

```
/**
```

```
* Returns a negative cycle reachable from the source vertex {@code s}, or
{@code null}
```

```
* if there is no such cycle.
```

```
* @return a negative cycle reachable from the source vertex {@code s}
```

```
* as an iterable of edges, and {@code null} if there is no such cycle
```

```
*/
```

```
public Iterable<DirectedEdge> negativeCycle() {
 return cycle;
}
```

```
// by finding a cycle in predecessor graph
```

```
private void findNegativeCycle() {
 int V = edgeTo.length;
 EdgeWeightedDigraph spt = new EdgeWeightedDigraph(V);
 for (int v = 0; v < V; v++)
 if (edgeTo[v] != null)
 spt.addEdge(edgeTo[v]);

 EdgeWeightedDirectedCycle finder = new EdgeWeightedDirectedCycle(spt);
 cycle = finder.cycle();
}
```

```
/**
```

```
 * Returns the length of a shortest path from the source vertex {@code s} to
 * vertex {@code v}.
```

```
 * @param v the destination vertex
```

```
 * @return the length of a shortest path from the source vertex {@code s} to
 * vertex {@code v};
```

```
 * {@code Double.POSITIVE_INFINITY} if no such path
```

```
 * @throws UnsupportedOperationException if there is a negative cost cycle
 * reachable
```

```
 * from the source vertex {@code s}
```

```

* @throws IllegalArgumentException unless {@code 0 <= v < V}
*/
public double distTo(int v) {
 validateVertex(v);
 if (hasNegativeCycle())
 throw new UnsupportedOperationException("Negative cost cycle exists");
 return distTo[v];
}

```

```

/**
 * Is there a path from the source {@code s} to vertex {@code v}?
 * @param v the destination vertex
 * @return {@code true} if there is a path from the source vertex
 * {@code s} to vertex {@code v}, and {@code false} otherwise
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public boolean hasPathTo(int v) {
 validateVertex(v);
 return distTo[v] < Double.POSITIVE_INFINITY;
}

```

```

/**
 * Returns a shortest path from the source {@code s} to vertex {@code v}.
 * @param v the destination vertex
 * @return a shortest path from the source {@code s} to vertex {@code v}
 * as an iterable of edges, and {@code null} if no such path

```

\* @throws UnsupportedOperationException if there is a negative cost cycle reachable

\* from the source vertex {@code s}

\* @throws IllegalArgumentException unless {@code 0 ≤ v < V}

\*/

```
public Iterable<DirectedEdge> pathTo(int v) {
 validateVertex(v);
 if (hasNegativeCycle())
 throw new UnsupportedOperationException("Negative cost cycle exists");
 if (!hasPathTo(v)) return null;
 Stack<DirectedEdge> path = new Stack<DirectedEdge>();
 for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
 path.push(e);
 }
 return path;
}
```

// check optimality conditions: either

// (i) there exists a negative cycle reachable from s

// or

// (ii) for all edges  $e = v \rightarrow w$ :  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$

// (ii') for all edges  $e = v \rightarrow w$  on the SPT:  $\text{distTo}[w] == \text{distTo}[v] + e.\text{weight}()$

private boolean check(EdgeWeightedDigraph G, int s) {

// has a negative cycle

if (hasNegativeCycle()) {

double weight = 0.0;

```

for (DirectedEdge e : negativeCycle()) {
 weight += e.weight();
}
if (weight >= 0.0) {
 System.err.println("error: weight of negative cycle = " + weight);
 return false;
}
}

// no negative cycle reachable from source
else {

 // check that distTo[v] and edgeTo[v] are consistent
 if (distTo[s] != 0.0 || edgeTo[s] != null) {
 System.err.println("distanceTo[s] and edgeTo[s] inconsistent");
 return false;
 }
 for (int v = 0; v < G.V(); v++) {
 if (v == s) continue;
 if (edgeTo[v] == null && distTo[v] != Double.POSITIVE_INFINITY) {
 System.err.println("distTo[] and edgeTo[] inconsistent");
 return false;
 }
 }
}

// check that all edges e = v->w satisfy distTo[w] <= distTo[v] + e.weight()

```

```

for (int v = 0; v < G.V(); v++) {
 for (DirectedEdge e : G.adj(v)) {
 int w = e.to();
 if (distTo[v] + e.weight() < distTo[w]) {
 System.err.println("edge " + e + " not relaxed");
 return false;
 }
 }
}

```

// check that all edges  $e = v \rightarrow w$  on SPT satisfy  $\text{distTo}[w] == \text{distTo}[v] + e.\text{weight}()$

```

for (int w = 0; w < G.V(); w++) {
 if (edgeTo[w] == null) continue;
 DirectedEdge e = edgeTo[w];
 int v = e.from();
 if (w != e.to()) return false;
 if (distTo[v] + e.weight() != distTo[w]) {
 System.err.println("edge " + e + " on shortest path not tight");
 return false;
 }
}
}

```

```

StdOut.println("Satisfies optimality conditions");
StdOut.println();
return true;

```

```
}
```

```
// throw an IllegalArgumentException unless {@code 0 <= v < V}
```

```
private void validateVertex(int v) {
```

```
 int V = distTo.length;
```

```
 if (v < 0 || v >= V)
```

```
 throw new IllegalArgumentException("vertex " + v + " is not between 0
and " + (V-1));
```

```
}
```

```
//the below function will convert positive value of currency to -log(value)
```

```
public static void convertLog(File file)
```

```
{
```

```
 try
```

```
{
```

```
 Scanner sc=new Scanner(file);
```

```
 FileWriter fw=new FileWriter("bin/CurrencyExchange.txt");
```

```
 String s;
```

```
 int i;
```

```
 int co=1;
```

```
 while(sc.hasNextLine())
```

```
{
```

```
 s=sc.nextLine();
```

```
 //System.out.println(s);
```

```
 if(co<=2)
```

```
{
```

```
 if(co==1)
```



```
{
size=Integer.parseInt(s);
}
fw.write(s+"\n");
}
else if(co>=3)
{
int pos=s.lastIndexOf(' ');
String s1="";
for(i=pos+1;i<s.length();i++)
{
s1=s1+s.charAt(i);
}
//System.out.println("String is : "+s1);
float n=(-1)*(float)Math.log((Float.parseFloat(s1)));
String s2=s.substring(0,pos+1)+String.valueOf(n)+"\n";
fw.write(s2);
}
co++;
}
fw.close();
sc.close();
}
catch(Exception e)
{
System.out.println(e);
```

```
}
```

```
//System.out.println("Completed !");
```

```
}
```

//the below function will read names from a file and store them in array 'name[]'

```
public static void readName(File file,String name[])
```

```
{
```

```
 try
```

```
 {
```

```
 Scanner sc=new Scanner(file);
```

```
 int i=0;
```

```
 while(sc.hasNextLine())
```

```
 {
```

```
 name[i]=sc.nextLine();
```

```
 i++;
```

```
 }
```

```
 sc.close();
```

```
 }
```

```
 catch(Exception e)
```

```
 {
```

```
 System.out.println(e);
```

```
 }
```

```
}
```

```
/**
```

\* Unit tests the {@code BellmanFordSP} data type.

\*

\* @param args the command-line arguments

\*/

```
public static void main(String[] args) {
 File file =new File("bin/"+args[0]);
 convertLog(file);
 File file2 =new File("bin/CurrencyExchange.txt");
 In in = new In(file2);
 int s = Integer.parseInt(args[1]);
 EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);

 BellmanFordSP sp = new BellmanFordSP(G, s);

 String name[]=new String[size];
 File file3=new File("bin/"+args[2]);
 readName(file3,name);

 // print negative cycle
 if (sp.hasNegativeCycle()) {
 double stake = 1000.0;
 for (DirectedEdge e : sp.negativeCycle())
 {
 StdOut.println(e);

 StdOut.printf("%10.5f %s ", stake, name[e.from()]);
 stake *= Math.exp(-e.weight());
 }
 }
}
```

```
StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
```

```
}
```

```
//StdOut.println(e);
```

```
}
```

```
// print shortest paths
```

```
else {
```

```
StdOut.printf("NO CAPITAL BENEFITTING, Shortest paths are as follows:
```

```
\n");
```

```
for (int v = 0; v < G.V(); v++) {
```

```
 if (sp.hasPathTo(v)) {
```

```
 StdOut.printf("%d to %d (%5.2f) ", s, v, sp.distTo(v));
```

```
 for (DirectedEdge e : sp.pathTo(v)) {
```

```
 StdOut.print(e + " ");
```

```
 }
```

```
 StdOut.println();
```

```
 }
```

```
else {
```

```
 StdOut.printf("%d to %d no path\n", s, v);
```

```
}
```

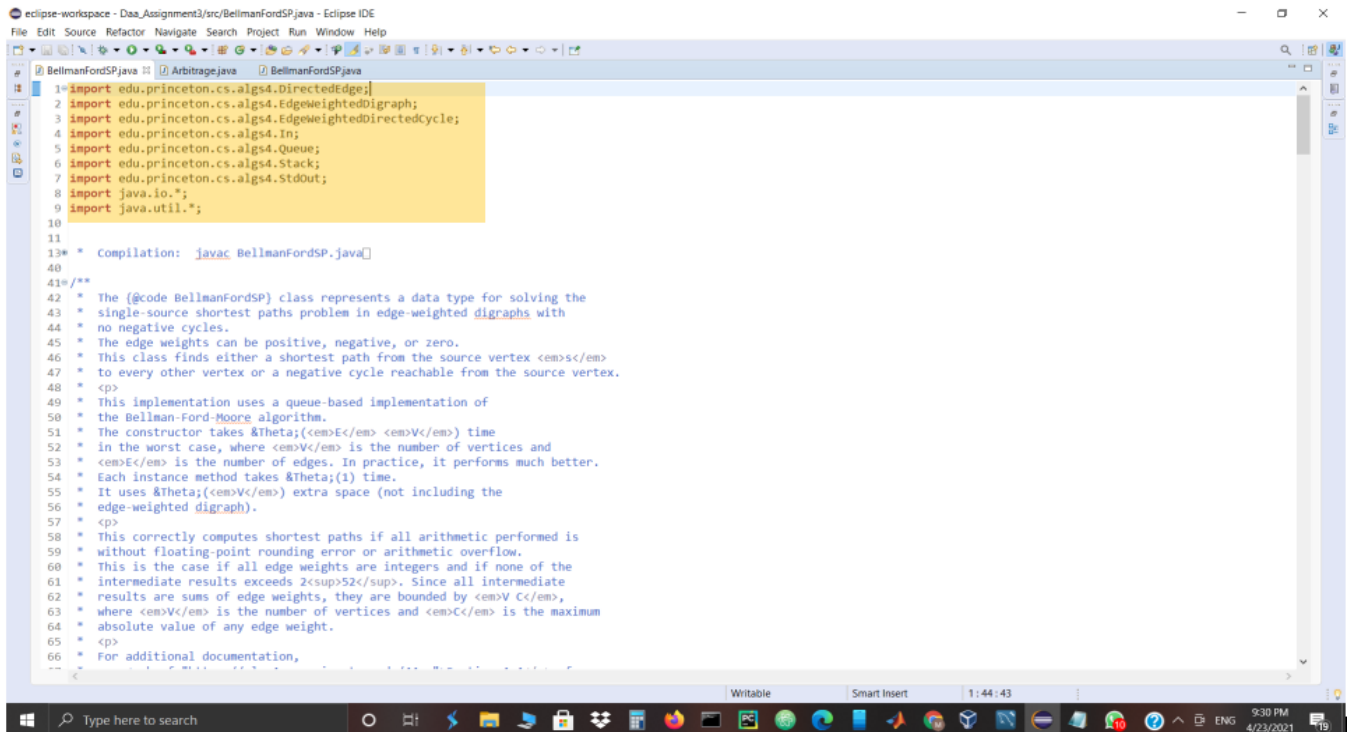
```
}
```

```
}
```

```
}
```

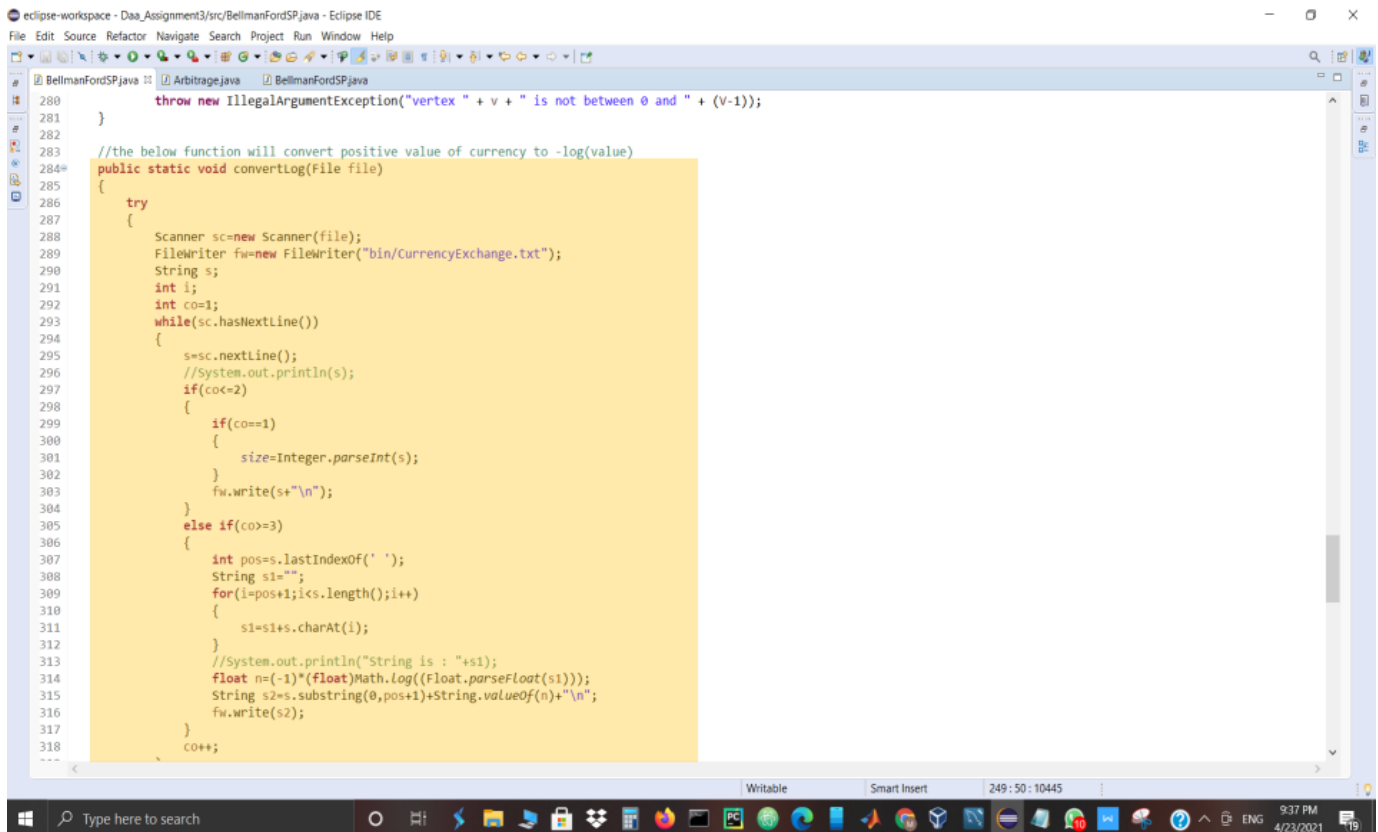
```
}
```

## Modifications in the code of BellmanFordSP is as follows (Highlighted Lines changed) :



```
1 import edu.princeton.cs.algs4.DirectedEdge;
2 import edu.princeton.cs.algs4.EdgeWeightedDigraph;
3 import edu.princeton.cs.algs4.EdgeWeightedDirectedCycle;
4 import edu.princeton.cs.algs4.In;
5 import edu.princeton.cs.algs4.Queue;
6 import edu.princeton.cs.algs4.Stack;
7 import edu.princeton.cs.algs4.StdOut;
8 import java.io.*;
9 import java.util.*;

10
11
12 * Compilation: javac BellmanFordSP.java
13
14
15 /**
16 * The {@code BellmanFordSP} class represents a data type for solving the
17 * single-source shortest paths problem in edge-weighted digraphs with
18 * no negative cycles.
19 * The edge weights can be positive, negative, or zero.
20 * This class finds either a shortest path from the source vertex s
21 * to every other vertex or a negative cycle reachable from the source vertex.
22 *
23 * <p>
24 * This implementation uses a queue-based implementation of
25 * the Bellman-Ford-Moore algorithm.
26 * The constructor takes Θ(V E) time
27 * in the worst case, where V is the number of vertices and
28 * E is the number of edges. In practice, it performs much better.
29 * Each instance method takes Θ(1) time.
30 * It uses Θ(V) extra space (not including the
31 * edge-weighted digraph).
32 *
33 * <p>
34 * This correctly computes shortest paths if all arithmetic performed is
35 * without floating-point rounding error or arithmetic overflow.
36 * This is the case if all edge weights are integers and if none of the
37 * intermediate results exceeds 2^{52} . Since all intermediate
38 * results are sums of edge weights, they are bounded by V C,
39 * where V is the number of vertices and C is the maximum
40 * absolute value of any edge weight.
41 *
42 * <p>
43 * For additional documentation,
44 * see the class documentation for
45 * EdgeWeightedDigraph,
46 * EdgeWeightedDirectedCycle,
47 * In,
48 * Queue,
49 * Stack,
50 * StdOut,
51 * In,
52 * Queue,
53 * Stack,
54 * StdOut,
55 * In,
56 * Queue,
57 * Stack,
58 * StdOut,
59 * In,
60 * Queue,
61 * Stack,
62 * StdOut,
63 * In,
64 * Queue,
65 * Stack,
66 * StdOut,
67 * In,
68 * Queue,
69 * Stack,
70 * StdOut,
71 * In,
72 * Queue,
73 * Stack,
74 * StdOut,
75 * In,
76 * Queue,
77 * Stack,
78 * StdOut,
79 * In,
80 * Queue,
81 * Stack,
82 * StdOut,
83 * In,
84 * Queue,
85 * Stack,
86 * StdOut,
87 * In,
88 * Queue,
89 * Stack,
90 * StdOut,
91 * In,
92 * Queue,
93 * Stack,
94 * StdOut,
95 * In,
96 * Queue,
97 * Stack,
98 * StdOut,
99 * In,
100 * Queue,
101 * Stack,
102 * StdOut,
103 * In,
104 * Queue,
105 * Stack,
106 * StdOut,
107 * In,
108 * Queue,
109 * Stack,
110 * StdOut,
111 * In,
112 * Queue,
113 * Stack,
114 * StdOut,
115 * In,
116 * Queue,
117 * Stack,
118 * StdOut,
119 * In,
120 * Queue,
121 * Stack,
122 * StdOut,
123 * In,
124 * Queue,
125 * Stack,
126 * StdOut,
127 * In,
128 * Queue,
129 * Stack,
130 * StdOut,
131 * In,
132 * Queue,
133 * Stack,
134 * StdOut,
135 * In,
136 * Queue,
137 * Stack,
138 * StdOut,
139 * In,
140 * Queue,
141 * Stack,
142 * StdOut,
143 * In,
144 * Queue,
145 * Stack,
146 * StdOut,
147 * In,
148 * Queue,
149 * Stack,
150 * StdOut,
151 * In,
152 * Queue,
153 * Stack,
154 * StdOut,
155 * In,
156 * Queue,
157 * Stack,
158 * StdOut,
159 * In,
160 * Queue,
161 * Stack,
162 * StdOut,
163 * In,
164 * Queue,
165 * Stack,
166 * StdOut,
167 * In,
168 * Queue,
169 * Stack,
170 * StdOut,
171 * In,
172 * Queue,
173 * Stack,
174 * StdOut,
175 * In,
176 * Queue,
177 * Stack,
178 * StdOut,
179 * In,
180 * Queue,
181 * Stack,
182 * StdOut,
183 * In,
184 * Queue,
185 * Stack,
186 * StdOut,
187 * In,
188 * Queue,
189 * Stack,
190 * StdOut,
191 * In,
192 * Queue,
193 * Stack,
194 * StdOut,
195 * In,
196 * Queue,
197 * Stack,
198 * StdOut,
199 * In,
200 * Queue,
201 * Stack,
202 * StdOut,
203 * In,
204 * Queue,
205 * Stack,
206 * StdOut,
207 * In,
208 * Queue,
209 * Stack,
210 * StdOut,
211 * In,
212 * Queue,
213 * Stack,
214 * StdOut,
215 * In,
216 * Queue,
217 * Stack,
218 * StdOut,
219 * In,
220 * Queue,
221 * Stack,
222 * StdOut,
223 * In,
224 * Queue,
225 * Stack,
226 * StdOut,
227 * In,
228 * Queue,
229 * Stack,
230 * StdOut,
231 * In,
232 * Queue,
233 * Stack,
234 * StdOut,
235 * In,
236 * Queue,
237 * Stack,
238 * StdOut,
239 * In,
240 * Queue,
241 * Stack,
242 * StdOut,
243 * In,
244 * Queue,
245 * Stack,
246 * StdOut,
247 * In,
248 * Queue,
249 * Stack,
250 * StdOut,
251 * In,
252 * Queue,
253 * Stack,
254 * StdOut,
255 * In,
256 * Queue,
257 * Stack,
258 * StdOut,
259 * In,
260 * Queue,
261 * Stack,
262 * StdOut,
263 * In,
264 * Queue,
265 * Stack,
266 * StdOut,
267 * In,
268 * Queue,
269 * Stack,
270 * StdOut,
271 * In,
272 * Queue,
273 * Stack,
274 * StdOut,
275 * In,
276 * Queue,
277 * Stack,
278 * StdOut,
279 * In,
280 * Queue,
281 * Stack,
282 * StdOut,
283 * In,
284 * Queue,
285 * Stack,
286 * StdOut,
287 * In,
288 * Queue,
289 * Stack,
290 * StdOut,
291 * In,
292 * Queue,
293 * Stack,
294 * StdOut,
295 * In,
296 * Queue,
297 * Stack,
298 * StdOut,
299 * In,
300 * Queue,
301 * Stack,
302 * StdOut,
303 * In,
304 * Queue,
305 * Stack,
306 * StdOut,
307 * In,
308 * Queue,
309 * Stack,
310 * StdOut,
311 * In,
312 * Queue,
313 * Stack,
314 * StdOut,
315 * In,
316 * Queue,
317 * Stack,
318 * StdOut,
319 * In,
320 * Queue,
321 * Stack,
322 * StdOut,
323 * In,
324 * Queue,
325 * Stack,
326 * StdOut,
327 * In,
328 * Queue,
329 * Stack,
330 * StdOut,
331 * In,
332 * Queue,
333 * Stack,
334 * StdOut,
335 * In,
336 * Queue,
337 * Stack,
338 * StdOut,
339 * In,
340 * Queue,
341 * Stack,
342 * StdOut,
343 * In,
344 * Queue,
345 * Stack,
346 * StdOut,
347 * In,
348 * Queue,
349 * Stack,
350 * StdOut,
351 * In,
352 * Queue,
353 * Stack,
354 * StdOut,
355 * In,
356 * Queue,
357 * Stack,
358 * StdOut,
359 * In,
360 * Queue,
361 * Stack,
362 * StdOut,
363 * In,
364 * Queue,
365 * Stack,
366 * StdOut,
367 * In,
368 * Queue,
369 * Stack,
370 * StdOut,
371 * In,
372 * Queue,
373 * Stack,
374 * StdOut,
375 * In,
376 * Queue,
377 * Stack,
378 * StdOut,
379 * In,
380 * Queue,
381 * Stack,
382 * StdOut,
383 * In,
384 * Queue,
385 * Stack,
386 * StdOut,
387 * In,
388 * Queue,
389 * Stack,
390 * StdOut,
391 * In,
392 * Queue,
393 * Stack,
394 * StdOut,
395 * In,
396 * Queue,
397 * Stack,
398 * StdOut,
399 * In,
400 * Queue,
401 * Stack,
402 * StdOut,
403 * In,
404 * Queue,
405 * Stack,
406 * StdOut,
407 * In,
408 * Queue,
409 * Stack,
410 * StdOut,
411 * In,
412 * Queue,
413 * Stack,
414 * StdOut,
415 * In,
416 * Queue,
417 * Stack,
418 * StdOut,
419 * In,
420 * Queue,
421 * Stack,
422 * StdOut,
423 * In,
424 * Queue,
425 * Stack,
426 * StdOut,
427 * In,
428 * Queue,
429 * Stack,
430 * StdOut,
431 * In,
432 * Queue,
433 * Stack,
434 * StdOut,
435 * In,
436 * Queue,
437 * Stack,
438 * StdOut,
439 * In,
440 * Queue,
441 * Stack,
442 * StdOut,
443 * In,
444 * Queue,
445 * Stack,
446 * StdOut,
447 * In,
448 * Queue,
449 * Stack,
450 * StdOut,
451 * In,
452 * Queue,
453 * Stack,
454 * StdOut,
455 * In,
456 * Queue,
457 * Stack,
458 * StdOut,
459 * In,
460 * Queue,
461 * Stack,
462 * StdOut,
463 * In,
464 * Queue,
465 * Stack,
466 * StdOut,
467 * In,
468 * Queue,
469 * Stack,
470 * StdOut,
471 * In,
472 * Queue,
473 * Stack,
474 * StdOut,
475 * In,
476 * Queue,
477 * Stack,
478 * StdOut,
479 * In,
480 * Queue,
481 * Stack,
482 * StdOut,
483 * In,
484 * Queue,
485 * Stack,
486 * StdOut,
487 * In,
488 * Queue,
489 * Stack,
490 * StdOut,
491 * In,
492 * Queue,
493 * Stack,
494 * StdOut,
495 * In,
496 * Queue,
497 * Stack,
498 * StdOut,
499 * In,
500 * Queue,
501 * Stack,
502 * StdOut,
503 * In,
504 * Queue,
505 * Stack,
506 * StdOut,
507 * In,
508 * Queue,
509 * Stack,
510 * StdOut,
511 * In,
512 * Queue,
513 * Stack,
514 * StdOut,
515 * In,
516 * Queue,
517 * Stack,
518 * StdOut,
519 * In,
520 * Queue,
521 * Stack,
522 * StdOut,
523 * In,
524 * Queue,
525 * Stack,
526 * StdOut,
527 * In,
528 * Queue,
529 * Stack,
530 * StdOut,
531 * In,
532 * Queue,
533 * Stack,
534 * StdOut,
535 * In,
536 * Queue,
537 * Stack,
538 * StdOut,
539 * In,
540 * Queue,
541 * Stack,
542 * StdOut,
543 * In,
544 * Queue,
545 * Stack,
546 * StdOut,
547 * In,
548 * Queue,
549 * Stack,
550 * StdOut,
551 * In,
552 * Queue,
553 * Stack,
554 * StdOut,
555 * In,
556 * Queue,
557 * Stack,
558 * StdOut,
559 * In,
560 * Queue,
561 * Stack,
562 * StdOut,
563 * In,
564 * Queue,
565 * Stack,
566 * StdOut,
567 * In,
568 * Queue,
569 * Stack,
570 * StdOut,
571 * In,
572 * Queue,
573 * Stack,
574 * StdOut,
575 * In,
576 * Queue,
577 * Stack,
578 * StdOut,
579 * In,
580 * Queue,
581 * Stack,
582 * StdOut,
583 * In,
584 * Queue,
585 * Stack,
586 * StdOut,
587 * In,
588 * Queue,
589 * Stack,
590 * StdOut,
591 * In,
592 * Queue,
593 * Stack,
594 * StdOut,
595 * In,
596 * Queue,
597 * Stack,
598 * StdOut,
599 * In,
600 * Queue,
601 * Stack,
602 * StdOut,
603 * In,
604 * Queue,
605 * Stack,
606 * StdOut,
607 * In,
608 * Queue,
609 * Stack,
610 * StdOut,
611 * In,
612 * Queue,
613 * Stack,
614 * StdOut,
615 * In,
616 * Queue,
617 * Stack,
618 * StdOut,
619 * In,
620 * Queue,
621 * Stack,
622 * StdOut,
623 * In,
624 * Queue,
625 * Stack,
626 * StdOut,
627 * In,
628 * Queue,
629 * Stack,
630 * StdOut,
631 * In,
632 * Queue,
633 * Stack,
634 * StdOut,
635 * In,
636 * Queue,
637 * Stack,
638 * StdOut,
639 * In,
640 * Queue,
641 * Stack,
642 * StdOut,
643 * In,
644 * Queue,
645 * Stack,
646 * StdOut,
647 * In,
648 * Queue,
649 * Stack,
650 * StdOut,
651 * In,
652 * Queue,
653 * Stack,
654 * StdOut,
655 * In,
656 * Queue,
657 * Stack,
658 * StdOut,
659 * In,
660 * Queue,
661 * Stack,
662 * StdOut,
663 * In,
664 * Queue,
665 * Stack,
666 * StdOut,
667 * In,
668 * Queue,
669 * Stack,
670 * StdOut,
671 * In,
672 * Queue,
673 * Stack,
674 * StdOut,
675 * In,
676 * Queue,
677 * Stack,
678 * StdOut,
679 * In,
680 * Queue,
681 * Stack,
682 * StdOut,
683 * In,
684 * Queue,
685 * Stack,
686 * StdOut,
687 * In,
688 * Queue,
689 * Stack,
690 * StdOut,
691 * In,
692 * Queue,
693 * Stack,
694 * StdOut,
695 * In,
696 * Queue,
697 * Stack,
698 * StdOut,
699 * In,
700 * Queue,
701 * Stack,
702 * StdOut,
703 * In,
704 * Queue,
705 * Stack,
706 * StdOut,
707 * In,
708 * Queue,
709 * Stack,
710 * StdOut,
711 * In,
712 * Queue,
713 * Stack,
714 * StdOut,
715 * In,
716 * Queue,
717 * Stack,
718 * StdOut,
719 * In,
720 * Queue,
721 * Stack,
722 * StdOut,
723 * In,
724 * Queue,
725 * Stack,
726 * StdOut,
727 * In,
728 * Queue,
729 * Stack,
730 * StdOut,
731 * In,
732 * Queue,
733 * Stack,
734 * StdOut,
735 * In,
736 * Queue,
737 * Stack,
738 * StdOut,
739 * In,
740 * Queue,
741 * Stack,
742 * StdOut,
743 * In,
744 * Queue,
745 * Stack,
746 * StdOut,
747 * In,
748 * Queue,
749 * Stack,
750 * StdOut,
751 * In,
752 * Queue,
753 * Stack,
754 * StdOut,
755 * In,
756 * Queue,
757 * Stack,
758 * StdOut,
759 * In,
760 * Queue,
761 * Stack,
762 * StdOut,
763 * In,
764 * Queue,
765 * Stack,
766 * StdOut,
767 * In,
768 * Queue,
769 * Stack,
770 * StdOut,
771 * In,
772 * Queue,
773 * Stack,
774 * StdOut,
775 * In,
776 * Queue,
777 * Stack,
778 * StdOut,
779 * In,
780 * Queue,
781 * Stack,
782 * StdOut,
783 * In,
784 * Queue,
785 * Stack,
786 * StdOut,
787 * In,
788 * Queue,
789 * Stack,
790 * StdOut,
791 * In,
792 * Queue,
793 * Stack,
794 * StdOut,
795 * In,
796 * Queue,
797 * Stack,
798 * StdOut,
799 * In,
800 * Queue,
801 * Stack,
802 * StdOut,
803 * In,
804 * Queue,
805 * Stack,
806 * StdOut,
807 * In,
808 * Queue,
809 * Stack,
810 * StdOut,
811 * In,
812 * Queue,
813 * Stack,
814 * StdOut,
815 * In,
816 * Queue,
817 * Stack,
818 * StdOut,
819 * In,
820 * Queue,
821 * Stack,
822 * StdOut,
823 * In,
824 * Queue,
825 * Stack,
826 * StdOut,
827 * In,
828 * Queue,
829 * Stack,
830 * StdOut,
831 * In,
832 * Queue,
833 * Stack,
834 * StdOut,
835 * In,
836 * Queue,
837 * Stack,
838 * StdOut,
839 * In,
840 * Queue,
841 * Stack,
842 * StdOut,
843 * In,
844 * Queue,
845 * Stack,
846 * StdOut,
847 * In,
848 * Queue,
849 * Stack,
850 * StdOut,
851 * In,
852 * Queue,
853 * Stack,
854 * StdOut,
855 * In,
856 * Queue,
857 * Stack,
858 * StdOut,
859 * In,
860 * Queue,
861 * Stack,
862 * StdOut,
863 * In,
864 * Queue,
865 * Stack,
866 * StdOut,
867 * In,
868 * Queue,
869 * Stack,
870 * StdOut,
871 * In,
872 * Queue,
873 * Stack,
874 * StdOut,
875 * In,
876 * Queue,
877 * Stack,
878 * StdOut,
879 * In,
880 * Queue,
881 * Stack,
882 * StdOut,
883 * In,
884 * Queue,
885 * Stack,
886 * StdOut,
887 * In,
888 * Queue,
889 * Stack,
890 * StdOut,
891 * In,
892 * Queue,
893 * Stack,
894 * StdOut,
895 * In,
896 * Queue,
897 * Stack,
898 * StdOut,
899 * In,
900 * Queue,
901 * Stack,
902 * StdOut,
903 * In,
904 * Queue,
905 * Stack,
906 * StdOut,
907 * In,
908 * Queue,
909 * Stack,
910 * StdOut,
911 * In,
912 * Queue,
913 * Stack,
914 * StdOut,
915 * In,
916 * Queue,
917 * Stack,
918 * StdOut,
919 * In,
920 * Queue,
921 * Stack,
922 *
```



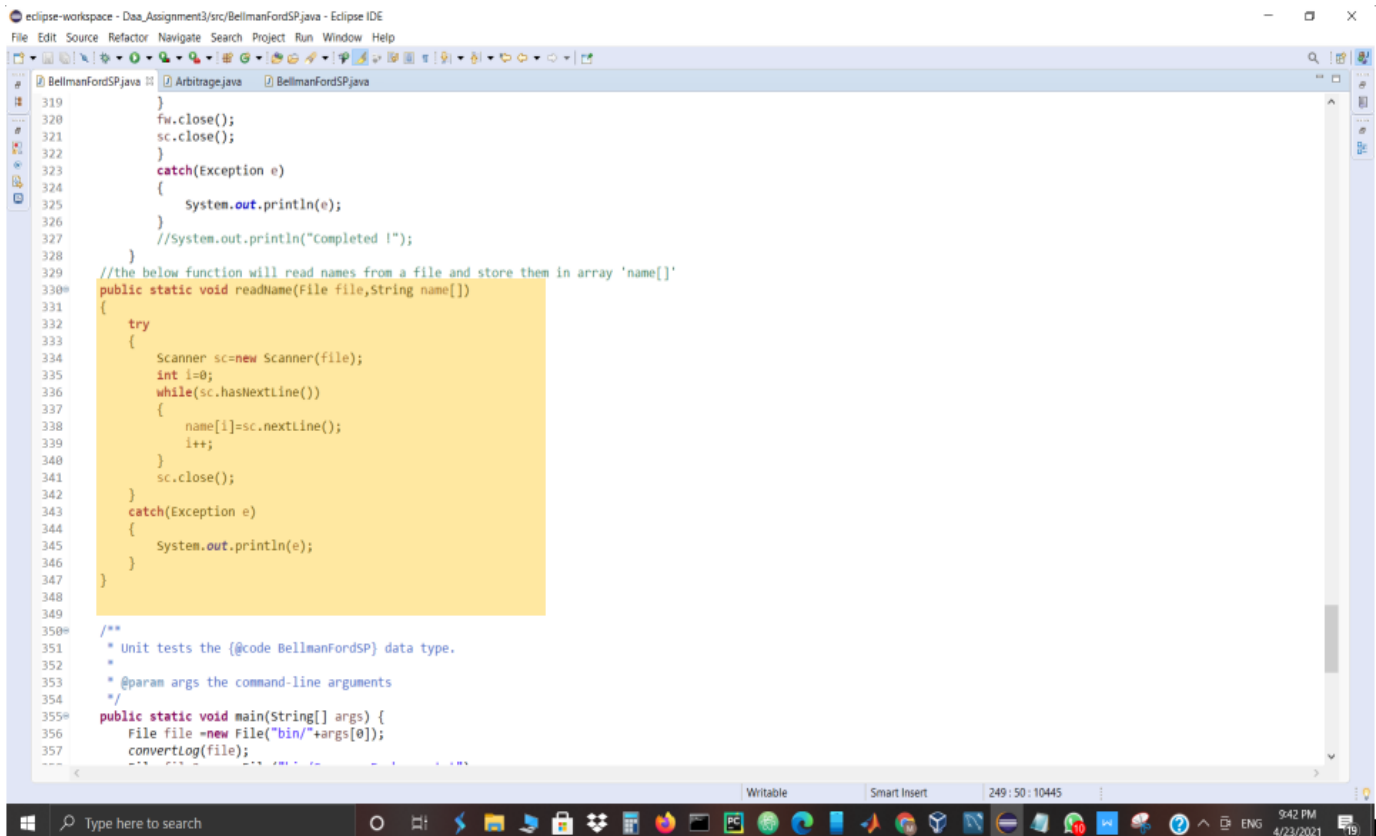
```
280 throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
281 }
282
283 //the below function will convert positive value of currency to -log(value)
284 public static void convertLog(File file)
285 {
286 try
287 {
288 Scanner sc=new Scanner(file);
289 FileWriter fw=new FileWriter("bin/CurrencyExchange.txt");
290 String s;
291 int i;
292 int co=1;
293 while(sc.hasNextLine())
294 {
295 s=sc.nextLine();
296 //System.out.println(s);
297 if(co<=2)
298 {
299 if(co==1)
300 {
301 size=Integer.parseInt(s);
302 }
303 fw.write(s+"\n");
304 }
305 else if(co>=3)
306 {
307 int pos=s.lastIndexOf(' ');
308 String s1="";
309 for(i=pos+1;i<s.length();i++)
310 {
311 s1=s1+s.charAt(i);
312 }
313 //System.out.println("String is : "+s1);
314 float n=(-1)*(float)Math.log((Float.parseFloat(s1)));
315 String s2=s.substring(0,pos+1)+String.valueOf(n)+"\n";
316 fw.write(s2);
317 }
318 co++;
319 }
320 }
321 }
322 }
```

Lines changed : Line 284 to 328

What is added : Added function **convertLog** so that it can convert the weights of currency exchange graph, where weights denote conversion rates to –  $\logarithm()$ . That is as follows :

$$\text{NewRate} = (-1) * \text{Math.log(OriginalRate)}$$

Reason to change : To calculate Capital benefitting by detecting **Negative cycles** in the currency exchange graph . So rate is changed to negative of its logarithm.

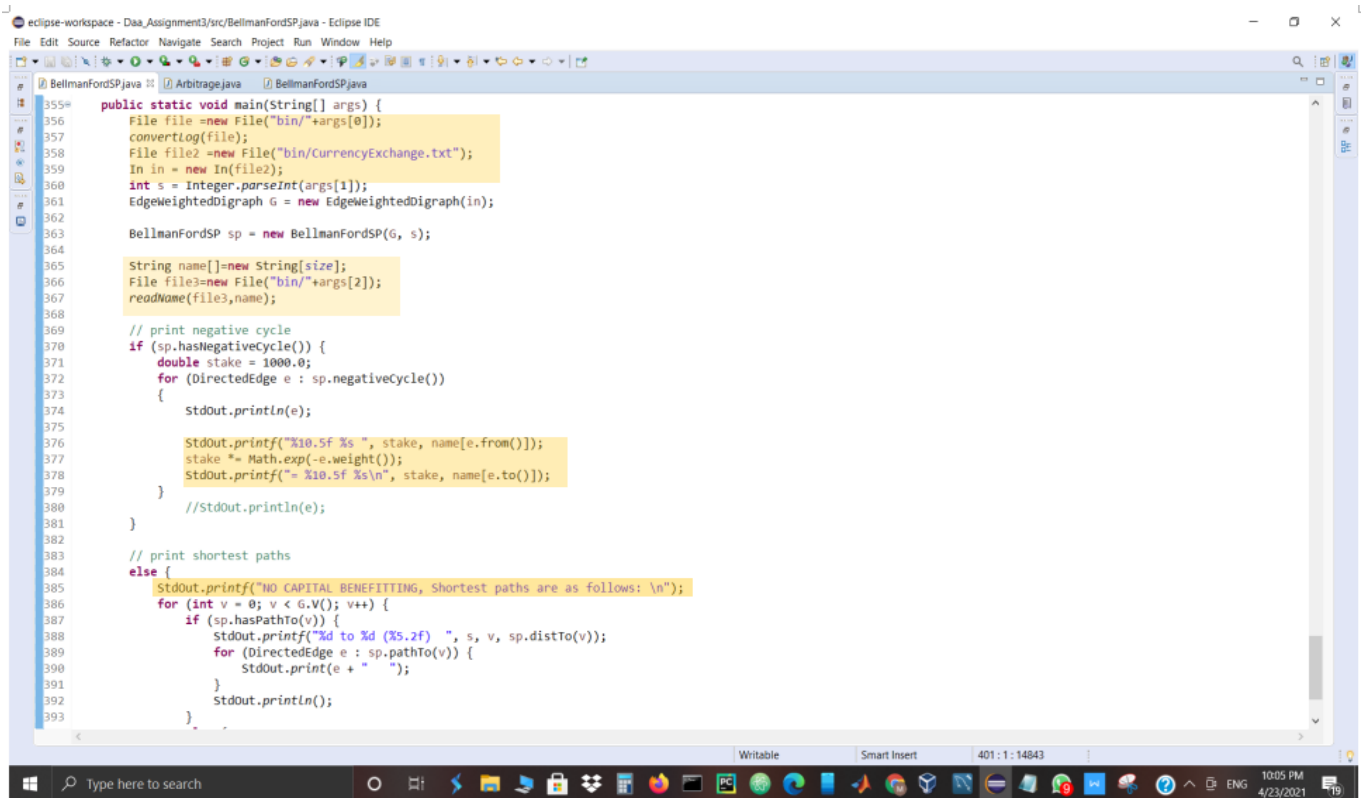


```
319 }
320 fw.close();
321 sc.close();
322 }
323 catch(Exception e)
324 {
325 System.out.println(e);
326 }
327 //System.out.println("Completed !");
328 }
329 //the below function will read names from a file and store them in array 'name[]'
330 public static void readName(File file,String name[])
331 {
332 try
333 {
334 Scanner sc=new Scanner(file);
335 int i=0;
336 while(sc.hasNextLine())
337 {
338 name[i]=sc.nextLine();
339 i++;
340 }
341 sc.close();
342 }
343 catch(Exception e)
344 {
345 System.out.println(e);
346 }
347 }
348
349
350 /**
351 * Unit tests the {@code BellmanFordSP} data type.
352 *
353 * @param args the command-line arguments
354 */
355 public static void main(String[] args) {
356 File file =new File("bin/"+args[0]);
357 convertLog(file);
358 }
```

Lines changed : Line 330 to 349

What is added : Added function **readName()** to read the names from a file named “names.txt” and store them in a array “name[]” for later use.

Reason to change : name[] array is used to demonstrate the execution of code for detection of negative cycles in **random currency exchange problem**.



```
355 public static void main(String[] args) {
356 File file = new File("bin/" + args[0]);
357 convertLog(file);
358 File file2 = new File("bin/CurrencyExchange.txt");
359 In in = new In(file2);
360 int s = Integer.parseInt(args[1]);
361 EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
362
363 BellmanFordSP sp = new BellmanFordSP(G, s);
364
365 String name[] = new String[size];
366 File file3 = new File("bin/" + args[2]);
367 readName(file3, name);
368
369 // print negative cycle
370 if (sp.hasNegativeCycle()) {
371 double stake = 1000.0;
372 for (DirectedEdge e : sp.negativeCycle())
373 {
374 StdOut.println(e);
375
376 StdOut.printf("%10.5f %s ", stake, name[e.from()]);
377 stake *= Math.exp(-e.weight());
378 StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
379 }
380 //StdOut.println(e);
381 }
382
383 // print shortest paths
384 else {
385 StdOut.printf("NO CAPITAL BENEFITTING, Shortest paths are as follows: \n");
386 for (int v = 0; v < G.V(); v++) {
387 if (sp.hasPathTo(v)) {
388 StdOut.printf("%d to %d (%5.2f) ", s, v, sp.distTo(v));
389 for (DirectedEdge e : sp.pathTo(v)) {
390 StdOut.print(e + " ");
391 }
392 StdOut.println();
393 }
394 }
395 }
396 }
```

Lines changed : Lines 356-359, Lines 376-378 and line 385

What is added : Called convertLog() function to convert rate to their  $(-1) \cdot \log()$  equivalent for calculation of capital benefit.

Saved the name of currencies in array called **name[]** and added print statements for printing of capital benefit or to detect if there is no capital benefit ( **NO NEGATIVE CYCLE** ).

Reason to change : convertLog() called to negate and take log of currency exchange rates for calculation of capital benefit and array declared and printing statements added just to “print the capital benefit for stack **1000** ” or check if no capital benefit possible.



**( b ) Demonstrate the random currency exchange table and its resulting graph :**

*(I) - The random currency exchange table is as follows :*

| <u>CURRENCIES</u> | <u>CONVERSION RATES</u> |               |               |                |               |
|-------------------|-------------------------|---------------|---------------|----------------|---------------|
| <i>RUP</i>        | <i>1</i>                | <i>0.0133</i> | <i>0.0862</i> | <i>1.0103</i>  | <i>0.0096</i> |
| <i>USD</i>        | <i>75.0778</i>          | <i>1</i>      | <i>6.48</i>   | <i>75.9925</i> | <i>0.7207</i> |
| <i>CNY</i>        | <i>11.56</i>            | <i>0.1540</i> | <i>1</i>      | <i>11.7081</i> | <i>0.1109</i> |
| <i>RUB</i>        | <i>0.9872</i>           | <i>0.0131</i> | <i>0.0853</i> | <i>1</i>       | <i>0.0095</i> |
| <i>GBP</i>        | <i>104.243</i>          | <i>1.388</i>  | <i>9.0084</i> | <i>105.513</i> | <i>1</i>      |

**Explanation of the above currency-exchange table :**

In the above currency exchange table , various symbols are as follows :

- RUP = Indian rupee
- USD = US dollar
- CNY = Chinese Yuan
- RUB = Russian rouble
- GBP = British Pound

The columns represent the **conversion rates** from one currency to another.

For ex:

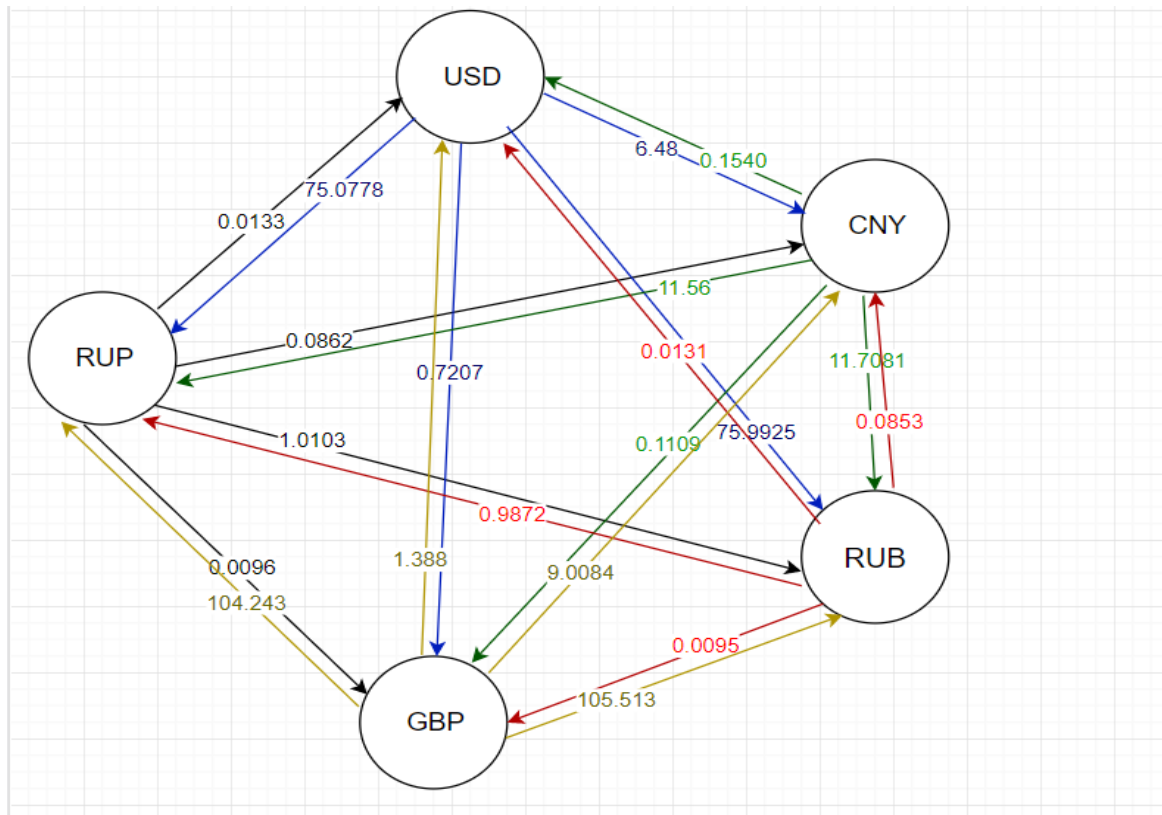
In third row :

- 1 CNY = 11.56 RUP
- 1 CNY = 0.1540 USD
- 1 CNY = 11.7081 RUB
- 1 CNY = 0.1109 GBP

Likewise the values in other rows are also defined as the example above.

This Table can be modelled in terms of a **Directed Graph** and **Bellman Ford algorithm** can be used to detect the negative cycles in the graph.

(II) - The resulting **Directed Graph** of the above table is as follows :



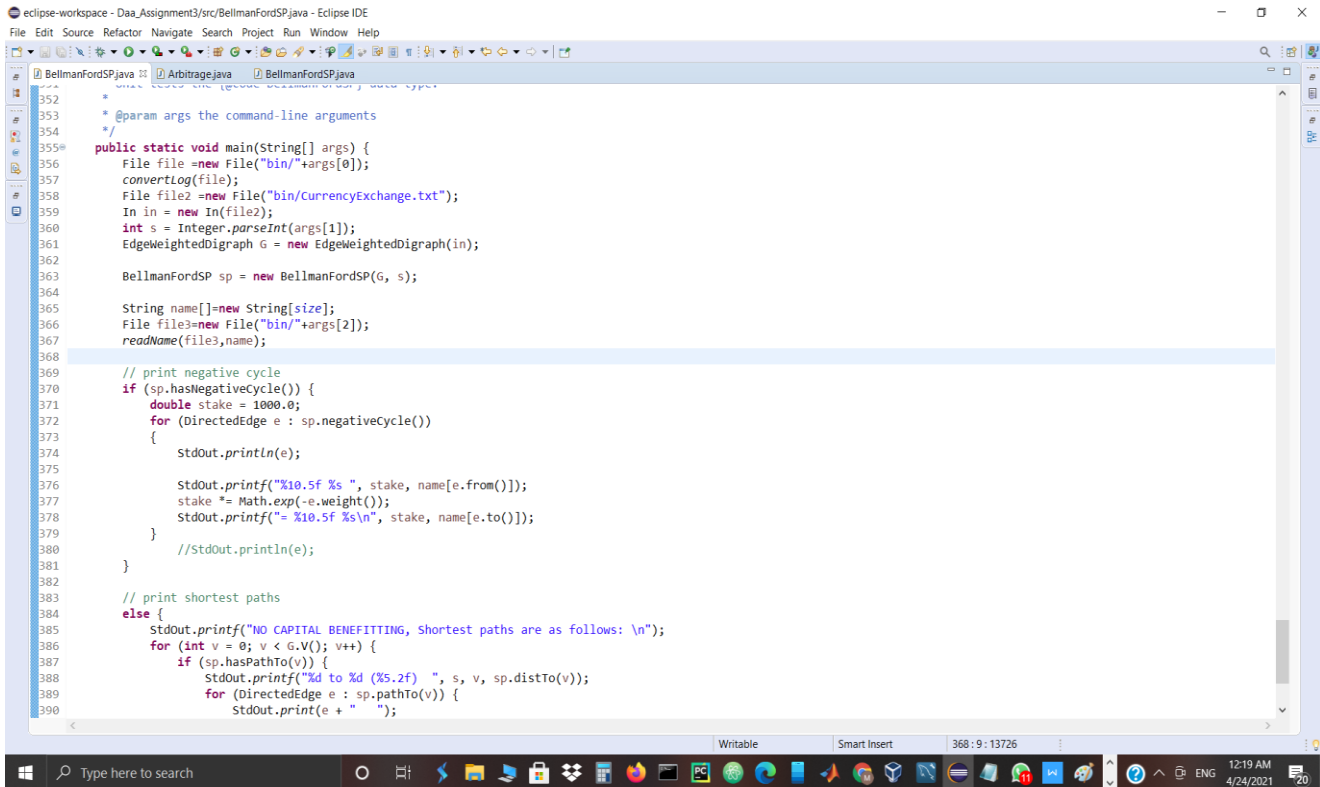
**Explanation of the above Currency-exchange Directed Graph :**

In the above directed graph , vertices represent the Different currencies ( RUP, USD, GBP , RUB , CNY ) . The edges represent the conversion from one currency to another .

The different colouring pattern is as follows :

- **BLACK COLOR** : for conversion rates from Rupees (RUP) to different currencies.
- **BLUE COLOR** : for conversion rates from US Dollars (USD) to different currencies.
- **GREEN COLOR** : for conversion rates from Chinese Yuan (CNY) to different currencies.
- **RED COLOR** : for conversion rates from Russian rouble ( RUB ) to different currencies.
- **YELLOW COLOR** : for conversion rates from British pound ( GBP ) to different currencies.

( c ) Demonstrate the execution of the code and on the above currency exchange to detect negative cycle :



```
352 *
353 * @param args the command-line arguments
354 */
355 public static void main(String[] args) {
356 File file = new File("bin/" + args[0]);
357 convertLog(file);
358 File file2 = new File("bin/CurrencyExchange.txt");
359 In in = new In(file2);
360 int s = Integer.parseInt(args[1]);
361 EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
362
363 BellmanFordSP sp = new BellmanFordSP(G, s);
364
365 String name[] = new String[size];
366 File file3 = new File("bin/" + args[2]);
367 readName(file3, name);
368
369 // print negative cycle
370 if (sp.hasNegativeCycle()) {
371 double stake = 1000.0;
372 for (DirectedEdge e : sp.negativeCycle())
373 {
374 Stdout.println(e);
375
376 Stdout.printf("%10.5f %s ", stake, name[e.from()]);
377 stake *= Math.exp(-e.weight());
378 Stdout.printf(" %10.5f %s\n", stake, name[e.to()]);
379 }
380 //Stdout.println(e);
381 }
382
383 // print shortest paths
384 else {
385 Stdout.printf("NO CAPITAL BENEFITTING, Shortest paths are as follows: \n");
386 for (int v = 0; v < G.V(); v++) {
387 if (sp.hasPathTo(v)) {
388 Stdout.printf("%d to %d (%5.2f) ", s, v, sp.distTo(v));
389 for (DirectedEdge e : sp.pathTo(v)) {
390 Stdout.print(e + " ");
391 }
392 }
393 }
394 }
395 }
```

Firstly , the main function of BellmanFordSP.java is being run . Its basic function is to read from two files :

- Names.txt
- CurrencyExchange.txt

Now a edgeWeightedGraph is created and BellmanFordSP Algo is called on the graph .

A if statement is included to check if the graph has negative cycles and print the capital benefit if found. If no capital benefit found , then print the message “Not found” and only print the Shortest path to each vertex from the source vertex.

```
eclipse-workspace - Daa_Assignment3/src/BellmanFordSP.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

BellmanFordSP.java
109
110 assert check(G, s);
111 }
112
113 // relax vertex v and put other endpoints on queue if changed
114 private void relax(EdgeWeightedDigraph G, int v) {
115 for (DirectedEdge e : G.adj(v)) {
116 int w = e.to();
117 if (distTo[w] > distTo[v] + e.weight() + EPSILON) {
118 distTo[w] = distTo[v] + e.weight();
119 edgeTo[w] = e;
120 if (!onQueue[w]) {
121 queue.enqueue(w);
122 onQueue[w] = true;
123 }
124 }
125 if (++cost % G.V() == 0) {
126 findNegativeCycle();
127 if (hasNegativeCycle()) return; // found a negative cycle
128 }
129 }
130 }
131
132 /**
133 * Is there a negative cycle reachable from the source vertex {@code s}?
134 * @return {@code true} if there is a negative cycle reachable from the
135 * source vertex {@code s}, and {@code false} otherwise
136 */
137 public boolean hasNegativeCycle() {
138 return cycle != null;
139 }
140
141 /**
142 * Returns a negative cycle reachable from the source vertex {@code s}, or {@code null}
143 * if there is no such cycle.
144 * @return a negative cycle reachable from the source vertex {@code s}
145 * as an Iterable of edges, and {@code null} if there is no such cycle
146 */
147 public Iterable<DirectedEdge> negativeCycle() {
```

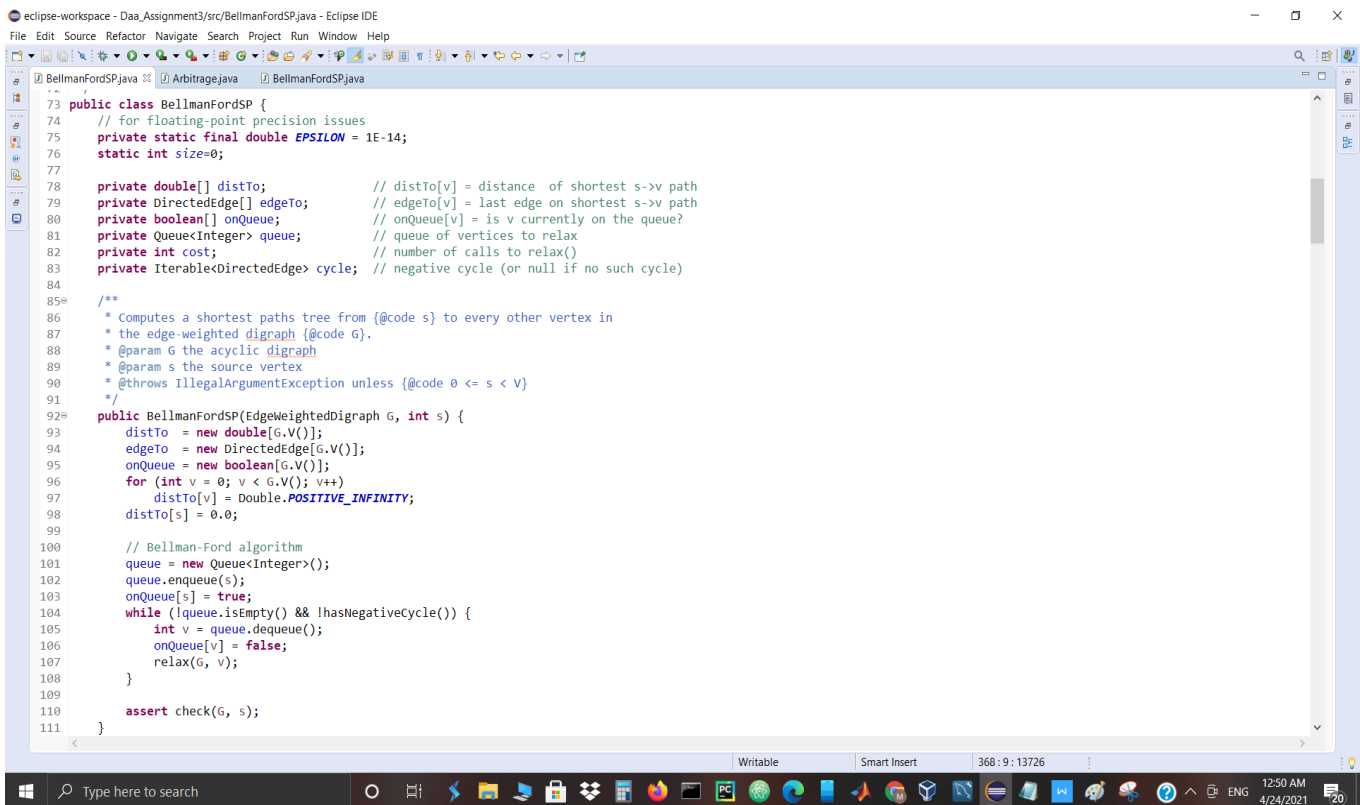
In the above code, there is relax function whose function is as follows :

It checks for the condition  $v.d > u.d + W(u,v)$  , where 'u' and 'v' are source and destination vertices and 'v.d' means distance to v from source and 'u.d' distance to u and  $W(u,v)$  is the weight of edge (u,v) .

If the condition is **TRUE** , then v.d is replace by " $u.d + W(u,v)$ " , and  $v.\pi = u$  , that is **PARENT OF v BECOMES u** .

This step is known as relaxation condition and is used in every Shortest-Path algorithm , and is the next step after the first step that is **INITIALIZATION**.

In this function , Negative cycle is also being checked by means of adding the weights of edges in a cycle and checking if its is less than 0 , if negative cycle is found , **hasNegativeCycle()** returns true.



```
73 public class BellmanFordSP {
74 // for floating-point precision issues
75 private static final double EPSILON = 1E-14;
76 static int size=0;
77
78 private double[] distTo; // distTo[v] = distance of shortest s->v path
79 private DirectedEdge[] edgeTo; // edgeTo[v] = last edge on shortest s->v path
80 private boolean[] onQueue; // onQueue[v] = is v currently on the queue?
81 private Queue<Integer> queue; // queue of vertices to relax
82 private int cost; // number of calls to relax()
83 private Iterable<DirectedEdge> cycle; // negative cycle (or null if no such cycle)
84
85 /**
86 * Computes a shortest paths tree from {@code s} to every other vertex in
87 * the edge-weighted digraph {@code G}.
88 * @param G the acyclic digraph
89 * @param s the source vertex
90 * @throws IllegalArgumentException unless {@code 0 <= s < V}
91 */
92 public BellmanFordSP(EdgeWeightedDigraph G, int s) {
93 distTo = new double[G.V()];
94 edgeTo = new DirectedEdge[G.V()];
95 onQueue = new boolean[G.V()];
96 for (int v = 0; v < G.V(); v++)
97 distTo[v] = Double.POSITIVE_INFINITY;
98 distTo[s] = 0.0;
99
100 // Bellman-Ford algorithm
101 queue = new Queue<Integer>();
102 queue.enqueue(s);
103 onQueue[s] = true;
104 while (!queue.isEmpty() && !hasNegativeCycle()) {
105 int v = queue.dequeue();
106 onQueue[v] = false;
107 relax(G, v);
108 }
109
110 assert check(G, s);
111 }
```

In the above image , constructor **BellmanFordSP()** is being defined.

Firstly , the initialization step is being done in which the starting vertex ( say 's') is assigned distance 0 and all other vertices are assigned distances  $+\infty$  . Now the queue is being initialized to keep track of the vertices that are being visited and a loop is run for  $|v|-1$  times for each edge , where  $|v|$  represents number of vertices .

During the running of the loop , if negative Cycle is being found, the loop is being terminated.

### **Demonstrating the working of code based on the currency exchange directed graph :**

The above currency-exchange directed graph contains 5 vertices and 20 edges . Now it is being passed to the BellmanFordSP.java for detection of negative cycle. Negative cycle is being detected and capital benefit is being calculated in terms of RUB and GBP.

Initially a stack of 1000 is maintained for calculation of capital benefit.

( d ) Write and describe the negative cycle in the currency exchange graph as well as in the code's output result (representing as capital benefit) :

For applying the above directed currency exchange graph , prepare a text file and name it as “curr1.txt” which contains the following information :

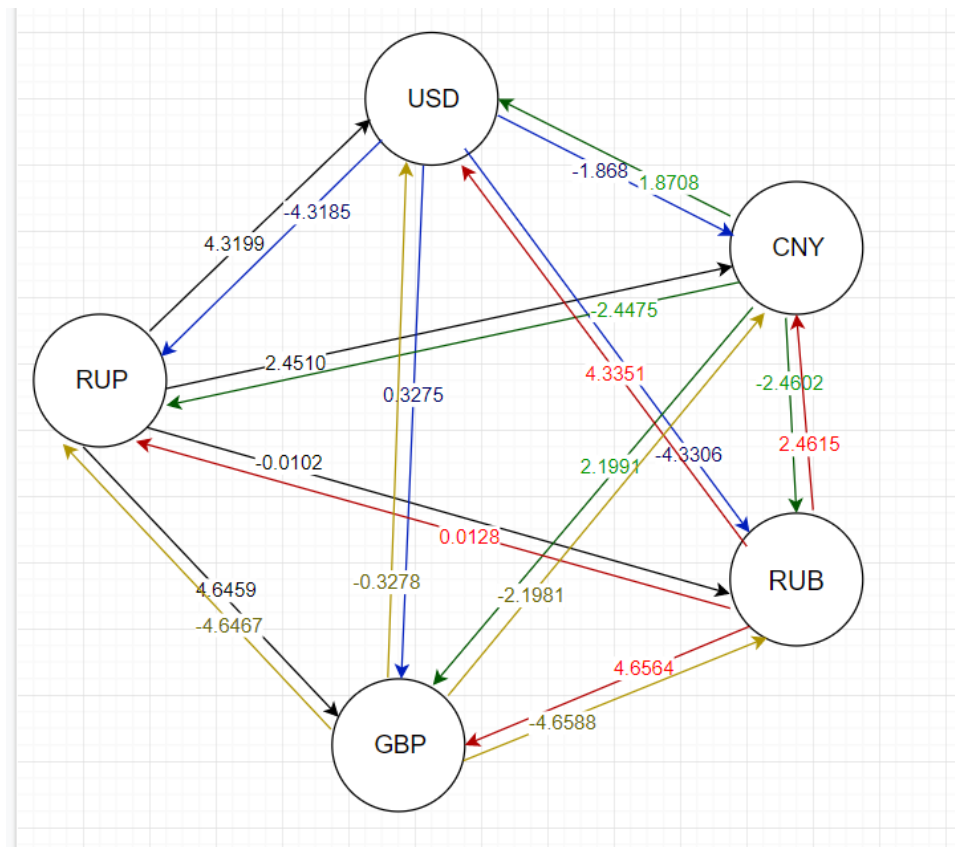
- First line contains **number of vertices = 5**
- Second line contains **number of edges = 20**
- Remaining 20 lines contains 3 values on each line < **source, end , weight**>

Also prepare a text file called as “names.txt” which contains the names of different currencies.

Now call the BellmanFordSP.java with three parameters :

```
% java BellmanFordSP curr1.txt 0 names.txt
```

Where 0 represents the initial source ( starting ) vertex.

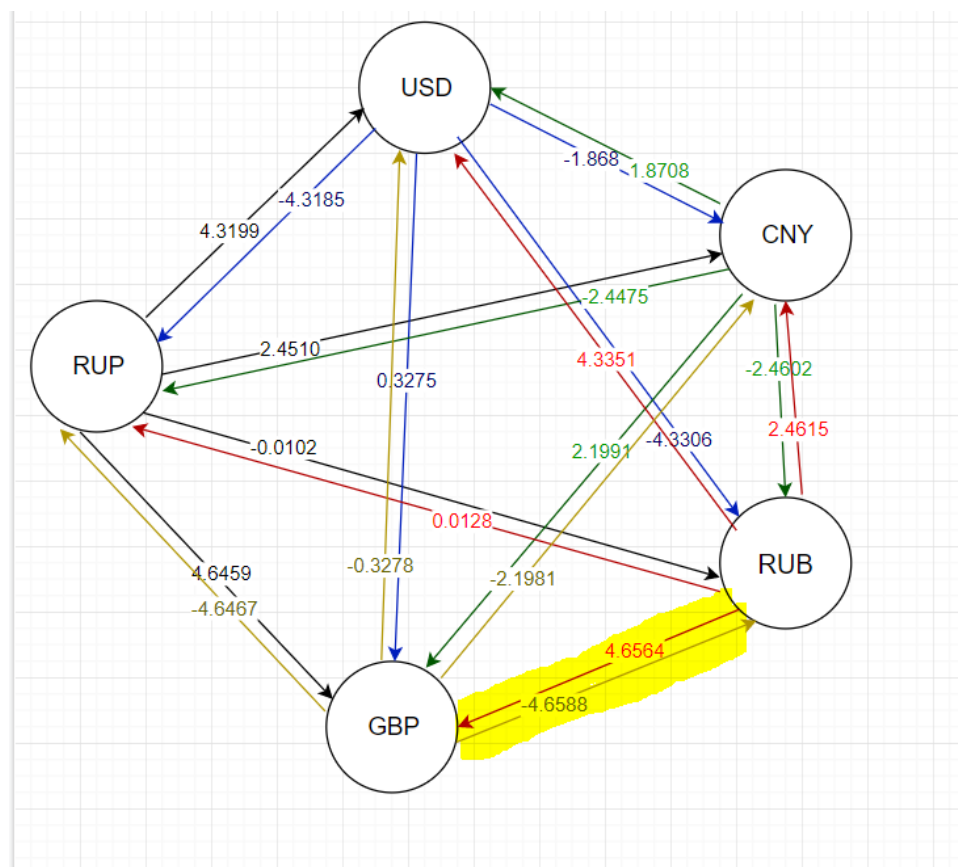


Before running the code, make sure that we represent each vertex from the directed graph above as a integer ranging from (0 to 4) for (RUP to GBP).

After making the text files and running the code through command listed above, following changes will be made to graph .

After applying the  $(-1) * \text{Math.log}(\text{rate})$  , the directed graph would look like as above.

Now the BellmanFordSP will run and find the negative cycle in the graph, fortunately we have found the negative cycle in the graph for two vertices GBP and RUB , which is highlighted in the next figure.



Now the negative cycle has been found , so we will proceed to the **if statement** in the code to calculate the capital benefit.

Now since a **Negative cycle** has been found , Capital benefit will be calculated by the formula written in code as shown in the figure below :

```
if (sp.hasNegativeCycle()) {
 double stake = 1000.0;
 for (DirectedEdge e : sp.negativeCycle())
 {
 StdOut.println(e);

 StdOut.printf("%10.5f %s ", stake, name[e.from()]);
 stake *= Math.exp(-e.weight());
 StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
 }
 //StdOut.println(e);
}
```

Now initially stake is taken to be 1000 ,

So if we calculate stake for edge RUB->GBP (3->4) , it is as follows :

$\text{stake} = \text{stake} * \text{Math.exp}(-4.6564)$

$\text{stake} = 9.50000$

So the first two lines in the output will be :

3->4 4.66

1000.00000 RUB = 9.50000 GBP



Now stake value is changed to 9.50000 , so for edge GBP->RUB (4->3) , it is as follows :

stake = stake \*Math.exp(4.6588)

stake = 1002.37321

So the last two lines in the output will be :

4->3 -4.66

9.50000 GBP = 1002.37321 RUB

Text file "curr1.txt" is as follows :

```
curr1 - Notepad
File Edit Format View Help
5
20
0 1 0.0133
0 2 0.0862
0 3 1.0103
0 4 0.0096
1 0 75.0778
1 2 6.48
1 3 75.9925
1 4 0.7207
2 0 11.56
2 1 0.1540
2 3 11.7081
2 4 0.1109
3 0 0.9872
3 1 0.0131
3 2 0.0853
3 4 0.0095
4 0 104.243
4 1 1.388
4 2 9.0084
4 3 105.513
```

Text file "names.txt" is as follows :

```
names - Notepad
File Edit Format View Help
RUB
USD
CNY
RUB
GBP
```

Final output from the code “BellmanFordSP.java” after passing the above two text files as an argument and 0 as the source vertex is as follows :

Console

<terminated> BellmanFordSP (1) [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe (Apr 24, 2021, 1:01:54 AM – 1:01:54 AM)

3->4 4.66

1000.00000 RUB = 9.50000 GBP

4->3 -4.66

9.50000 GBP = 1002.37321 RUB

The output is discussed above and here numbers 3 and 4 represent the currencies RUB and GBP.

Now capital benefit is as follows :

1000 RUB gets converted to 9.5 GBP and after some time 9.5 GBP gets converted back to 1002.37321 RUB.

So the profit achieved (capital benefit) is :  $1002.37321 - 1000 = 2.37321$  RUB

## CONCLUSION

This assignment helped me to learn the topics of Graph theory which are :

- Acyclic Longest Path Calculation
- BellmanFordSP Algorithm

I also learnt the application of these basics of graph theory to two main problems:

- Parallel Job scheduling problem making use of AcyclicLP algorithm
- Currency Exchange problem making use of BellmanFordSP algorithm