Lab 17

CSE 274

I. The Vertex class

The `Vertex` class is as follows:

```
class Vertex
{
      String label;

      Vertex(String theLabel)
      {
      label = theLabel;
      }
}
```

An object of a `Vertex` class is referred to as vertex.

II. The Node class

The `Node` class is as follows:

```
class Node {
      Vertex myVertex;
      int value;
      Node down;
      Node up;

      Node(Vertex myVertex, int value) {
            this.value = value;
            this.myVertex=myVertex;
            up = null;
            down = null;
      }

      public int getValue() {
            return value;
      }

      public Node getdown() {
            return down;
      }

      public Node getup() {
            return up;
      }
}
```

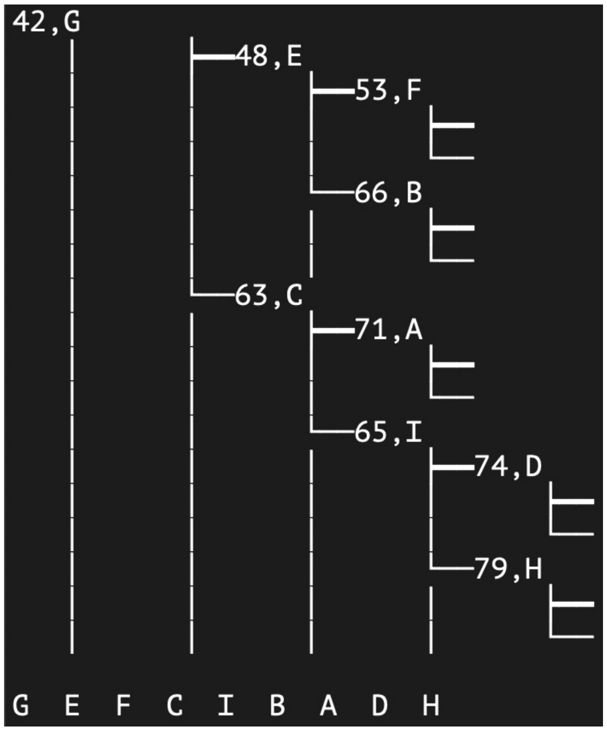
An object of `Node` class is referred to as a node. From above, every node has a `myVertex` variable which is a reference to a vertex.

III. THE MinHeap CLASS

The `MinHeap` class creates a Min Heap data structure based on trees. The Heap stores nodes, i.e. objects of `Node` class. New nodes are placed in the Heap according to the `value` variables of the nodes. The `delete` method of the `MinHeap` class `return` the `myVertex` variable of the root node of the Heap. Use the following lines of code to test the `delete` method of `MinHeap` class:

```
Vertex VertexA=new Vertex("A");
Vertex VertexB=new Vertex("B");
Vertex VertexC=new Vertex("C");
Vertex VertexD=new Vertex("D");
Vertex VertexE=new Vertex("E");
Vertex VertexF=new Vertex("F");
Vertex VertexG=new Vertex("G");
Vertex VertexH=new Vertex("H");
Vertex VertexI=new Vertex("I");
MinHeap myHeap= new MinHeap();
myHeap.add(VertexA, 71);
myHeap.add(VertexB, 66);
myHeap.add(VertexC, 63);
myHeap.add(VertexD, 74);
myHeap.add(VertexE, 48);
myHeap.add(VertexF, 53);
myHeap.add(VertexG, 42);
myHeap.add(VertexH, 79);
myHeap.add(VertexI, 65);
myHeap.display();
while(!myHeap.isEmpty())
      System.out.print(myHeap.delete().label+" ");
System.out.println("");
```

The expected output is printed below:

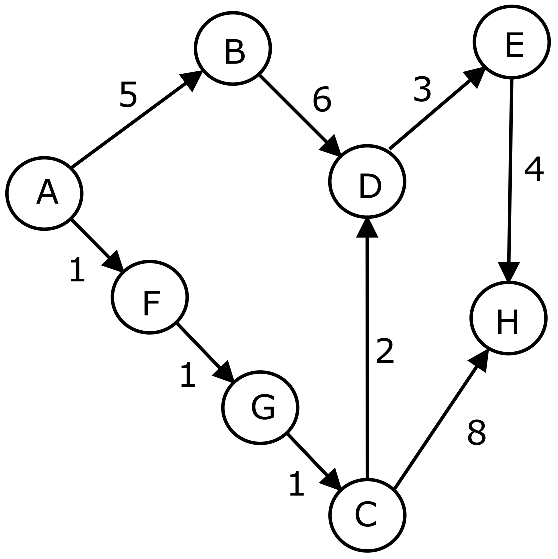


```
42,G
   │      ├──48,E
   │      │      ├──53,F
   │      │      │      ├──
   │      │      │      └──
   │      │      └──66,B
   │      │      │      ├──
   │      │      │      └──
   │      └──63,C
   │      │      ├──71,A
   │      │      │      ├──
   │      │      │      └──
   │      │      └──65,I
   │      │      │      ├──74,D
   │      │      │      │      ├──
   │      │      │      │      └──
   │      │      │      └──79,H
   │      │      │      │      ├──
   │      │      │      │      └──

G  E  F  C  I  B  A  D  H
```

IV. Directed Weighted Graphs

In a directed weighted graph, each edge of the graph has a weight and a direction. For instance, in the below directed weighted graph the edge between vertex `A` and vertex `B` has a weight of 5. Also, the direction of this edge is from vertex `A` to vertex `B`.



A directed weighted graph can be implemented using an Adjacency Matrix. Let say that the vertex `A` has an index of `i` and the vertex `B` has an index of `j` in a `VertexArray` that contains all vertexes of the directed weighted graph. Then:

- If there is an edge from vertex `A` to vertex `B`, the entry `Adjacency[i][j]` is the weight of this edge.
- If there is no edge from vertex `A` to vertex `B`, the entry `Adjacency[i][j]` is `-1`.

The `WeightedGraph` class is as follows:

```
class WeightedGraph
{
      private int[][] Adjacency;
      private Vertex[] VertexArray;

      int size;

      WeightedGraph() {
            size = 0;
}
```

The `VertexArray` is an array that stores the vertexes of the graph. The `size` is the number of vertexes in the graph. A new vertex can be added to the graph by using the `addVertex` method, which is already developed in the `Application.java` file. When calling this method to add a new node to the graph, the method initializes the new row and the new column of the `Adjacency` matrix to `-1`.

The following `addWeightedEdge` method is already developed for the `WeightedGraph` class:

```
void addWeightedEdge(String SLabel, String DLabel, int weight)
```

The above method receives a `SLabel` as the label of a source vertex, a `DLabel` as the label of a destination vertex, and a `weight` for the edge from the source vertex to the destination vertex. The method creates the said edge with the given weight by adjusting the `Adjacency` matrix. Use the following lines of code to test the developed method:

```
WeightedGraph myGraph = new WeightedGraph();
myGraph.addVertex("A");
myGraph.addVertex("B");
myGraph.addVertex("C");
myGraph.addVertex("D");
myGraph.addVertex("E");
myGraph.addVertex("F");
myGraph.addVertex("G");
myGraph.addVertex("H");
myGraph.addWeightedEdge("A", "B", 5);
myGraph.addWeightedEdge("B", "D", 6);
myGraph.addWeightedEdge("D", "E", 3);
myGraph.addWeightedEdge("C", "D", 2);
myGraph.addWeightedEdge("A", "F", 1);
myGraph.addWeightedEdge("F", "G", 1);
myGraph.addWeightedEdge("G", "C", 1);
myGraph.addWeightedEdge("E", "H", 4);
myGraph.addWeightedEdge("C", "H", 8);
myGraph.display();
```

The expected output is printed below:

```
Adjacency Matrix:

       A       B       C       D       E       F       G       H
A      -1      5       -1      -1      -1      1       -1      -1
B      -1      -1      -1      6       -1      -1      -1      -1
C      -1      -1      -1      2       -1      -1      -1      8
D      -1      -1      -1      -1      3       -1      -1      -1
E      -1      -1      -1      -1      -1      -1      -1      4
F      -1      -1      -1      -1      -1      -1      1       -1
G      -1      -1      1       -1      -1      -1      -1      -1
H      -1      -1      -1      -1      -1      -1      -1      -1
```

V. The `FindDistance` Method

In a directed weighted graph, the length of a path from a source vertex to a destination vertex is the summation of the weights of the specific edges that are part of the path. For instance, in the directed weighted graph of section IV, A-B-D-E-H is a path from vertex `A` to vertex `H`. This path includes the following edges: A-B, B-D, D-E and E-H. Since the weights of these edges are 5, 6, 3 and 4, the length of the said path is 5+6+3+4=18. In a directed weighted graph there might be several paths from a source vertex to a destination vertex. For instance, in the directed weighted graph of section IV, A-F-G-C-H is another path from vertex `A` to vertex `H`. This path has a length of 1+1+1+8=11.

The shortest path from a source vertex to a destination vertex is the one that has the shortest length. In the directed weighted graph of section IV, the shortest path from vertex `A` to vertex `H` is A-F-G-C-H. The length of the shortest path from a source vertex to a destination vertex is referred to as the distance from the source vertex to the destination vertex. Accordingly, in the directed weighted graph of section IV the distance from vertex `A` to vertex `H` is 11.

Develop the following method for the `WeightedGraph` class:

```
public void FindDistance(String SLabel)
```

The above method receives a `SLabel` as the label of a source vertex and prints the distance from the source vertex for other vertexes of the graph. Use the following logic in developing the method:

```
int[] distance = new int[size]
Vertex CurrentVertex;

for (int i = 0; i < size; i++)
   distance[i] = 1000

MinHeap myHeap = new MinHeap()
add the vertex with SLabel to myHeap
distance[index of the vertex with SLabel] = 0

while myHeap is not Empty
   CurrentVertex = myHeap.delete()

   for (int j = 0; j < size; j++)
            if CurrentVertex is neighbor with vertex VertexArray[j]

                  Vertex NeighborVertex = VertexArray[j]

                  int NewDistance =
                  distance[VertexToIndex(CurrentVertex)] +
                  weight of edge from CurrentVertex to
                  NeighborVertex

                  if (NewDistance < distance[j])
                        distance[j] = NewDistance
                        myHeap.add(NeighborVertex, distance[j])

System.out.println("Distance from vertex " + SLabel + " to vertex")

for (int i = 0; i < size; i++)
   System.out.print(VertexArray[i].label+ ": "+distance[i]+"\n")

return
```

In the above logic, `distance` is an array that stores distances from the source vertex to other vertexes of the graph. Initially, the `distance` array is set to 0 at the index corresponding to the source vertex, and is set to infinity, i.e. a very large number such as 1000, at the indexes corresponding to other vertexes. The `distance` array is updated in iterations of a `while` loop. In each iteration a `CurrentVertex` is taken/deleted from the heap. The `distance` from the source vertex to the neighbors of the `CurrentVertex` is updated if the `CurrentVertex` provides a shorter path to its neighbors compared to the paths that are already checked in previous iterations of the `while` loop. Those neighbors of `CurrentVertex` which have an updated `distance` are added to the heap as they might provide shorter paths to their own neighbors in the next iterations of the `while` loop. When placing the vertexes in the heap, their updated `distance`-es form the source vertex is used as their priority in the heap. Accordingly, in each iteration the vertex with the smallest `distance` from the source vertex is taken/deleted from the heap to be used as the `CurrentVertex`. This is required for the algorithm to work correctly. Use the following lines of code to test the developed method:

```
WeightedGraph myGraph = new WeightedGraph();
myGraph.addVertex("A");
myGraph.addVertex("B");
myGraph.addVertex("C");
myGraph.addVertex("D");
myGraph.addVertex("E");
myGraph.addVertex("F");
myGraph.addVertex("G");
myGraph.addVertex("H");
myGraph.addWeightedEdge("A", "B", 5);
myGraph.addWeightedEdge("B", "D", 6);
myGraph.addWeightedEdge("D", "E", 3);
myGraph.addWeightedEdge("C", "D", 2);
myGraph.addWeightedEdge("A", "F", 1);
myGraph.addWeightedEdge("F", "G", 1);
myGraph.addWeightedEdge("G", "C", 1);
myGraph.addWeightedEdge("E", "H", 4);
myGraph.addWeightedEdge("C", "H", 8);
myGraph.FindDistance("A");
```

The expected output is printed below:

```
Distance from vertex A to vertex
A: 0
B: 5
C: 3
D: 5
E: 8
F: 1
G: 2
H: 11
```

VI. The FindShortestPath method

Develop the following method for the `WeightedGraph` class:

```
public void FindShortestPath(String SLabel,String DLabel)
```

The above method receives a `SLabel` and a `DLabel` as the labels of a source and a destination vertex, and provides the shortest path from the source label to the destination label. The `FindShortestPath` method implements the same logic used in `FindDistance` method. Namely, the `FindShortestPath` method keeps updating a `distance` array in iterations of a `while` loop. The `FindShortestPath` method updates also a `PreviousVertexArray` in these iterations. More precisely, when `distance[j]` is updated in an iteration, the `PreviousVertexArray[j]` is updated to `CurrentVertex`, where the `CurrentVertex` is the specific vertex that is taken/deleted from the heap in the iteration. Initially, the `PreviousVertexArray` is set to 0 for the index corresponding to the source label and is set to infinity, i.e. a large number such as 1000, for other indexes. After completion of execution of the `while` loop, for each index `j` the variable `PreviousVertexArray[j]` shows the previous vertex of `VertexArray[j]` in the shortest path from the source vertex to the `VertexArray[j]`. Accordingly, the following lines of code at the end of the `FindShortestPath` method print the shortest path from the source vertex to the destination vertex:

```
System.out.print("Shortest Path from "+SLabel+" to "+DLabel+": ");
String ShortestPath="";
CurrentVertex=LabelToVertex(DLabel);
while (!CurrentVertex.label.equals(SLabel)) {
      int Index=VertexToIndex(CurrentVertex);
      CurrentVertex=PreviousVertexArray[Index];
      ShortestPath=CurrentVertex.label+"-"+ShortestPath;}
ShortestPath=ShortestPath+DLabel;
System.out.println(ShortestPath);
```

Use the following lines of code to test the developed method:

```
WeightedGraph myGraph = new WeightedGraph();
myGraph.addVertex("A");
myGraph.addVertex("B");
myGraph.addVertex("C");
myGraph.addVertex("D");
myGraph.addVertex("E");
myGraph.addVertex("F");
myGraph.addVertex("G");
myGraph.addVertex("H");
myGraph.addWeightedEdge("A", "B", 5);
myGraph.addWeightedEdge("B", "D", 6);
myGraph.addWeightedEdge("D", "E", 3);
myGraph.addWeightedEdge("C", "D", 2);
myGraph.addWeightedEdge("A", "F", 1);
myGraph.addWeightedEdge("F", "G", 1);
myGraph.addWeightedEdge("G", "C", 1);
myGraph.addWeightedEdge("E", "H", 4);
myGraph.addWeightedEdge("C", "H", 8);
myGraph.FindShortestPath("A", "H");
myGraph.FindShortestPath("A", "B");
myGraph.FindShortestPath("C", "E");
myGraph.FindShortestPath("F", "D");
myGraph.FindShortestPath("C", "H");
```

The expected output is printed below:

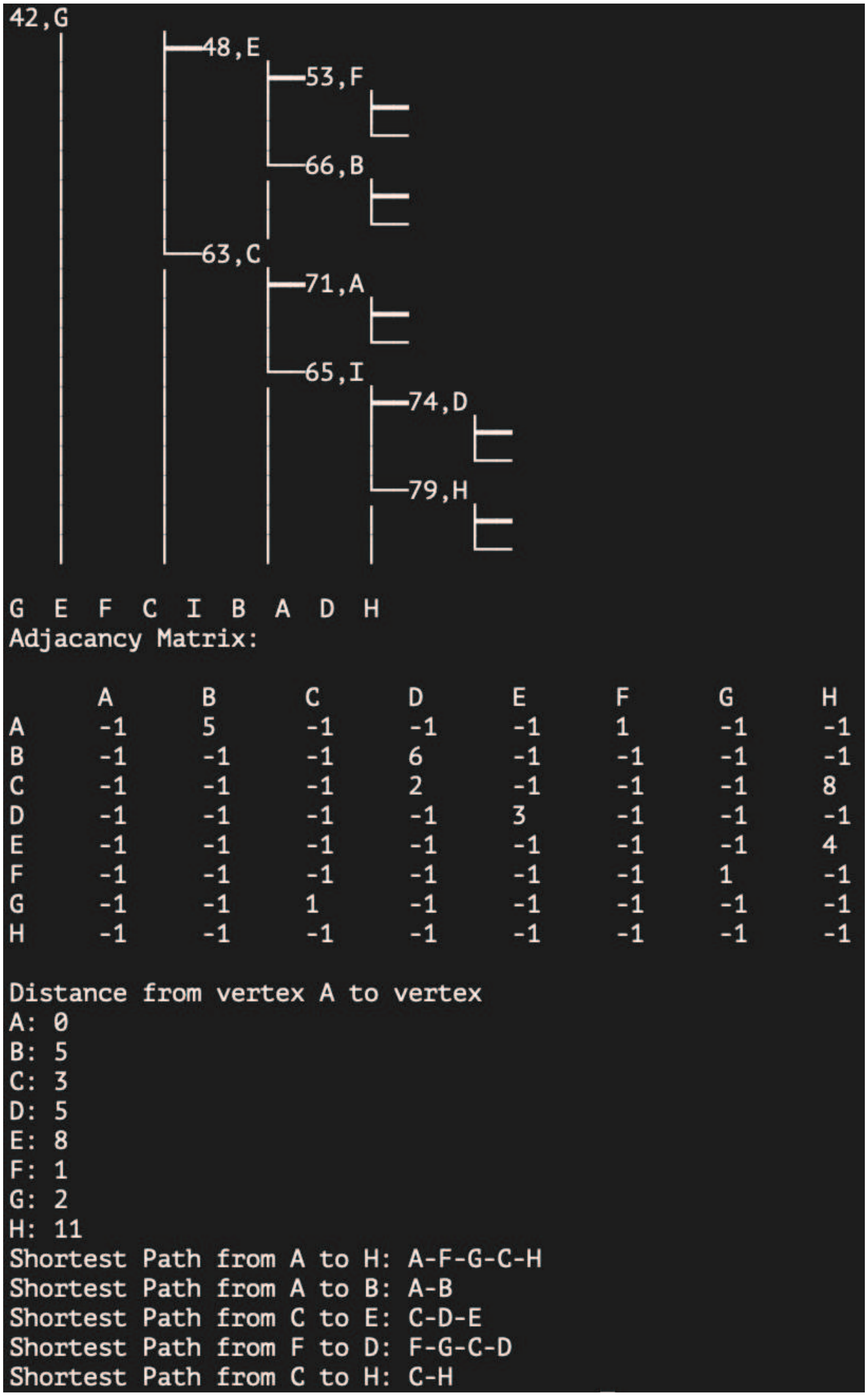
```
Shortest Path from A to H: A-F-G-C-H
Shortest Path from A to B: A-B
Shortest Path from C to E: C-D-E
Shortest Path from F to D: F-G-C-D
Shortest Path from C to H: C-H
```

VII. Submitting the Assignment

When submitting your response to this assignment, copy the following lines of code into the body of the `main` method:

```
///////// Test 1
Vertex VertexA=new Vertex("A");
Vertex VertexB=new Vertex("B");
Vertex VertexC=new Vertex("C");
Vertex VertexD=new Vertex("D");
Vertex VertexE=new Vertex("E");
Vertex VertexF=new Vertex("F");
Vertex VertexG=new Vertex("G");
Vertex VertexH=new Vertex("H");
Vertex VertexI=new Vertex("I");
MinHeap myHeap= new MinHeap();
myHeap.add(VertexA, 71);
myHeap.add(VertexB, 66);
myHeap.add(VertexC, 63);
myHeap.add(VertexD, 74);
myHeap.add(VertexE, 48);
myHeap.add(VertexF, 53);
myHeap.add(VertexG, 42);
myHeap.add(VertexH, 79);
myHeap.add(VertexI, 65);
myHeap.display();
while(!myHeap.isEmpty())
      System.out.print(myHeap.delete().label+" ");
System.out.println("");
///////// Test 2
WeightedGraph myGraph = new WeightedGraph();
myGraph.addVertex("A");
myGraph.addVertex("B");
myGraph.addVertex("C");
myGraph.addVertex("D");
myGraph.addVertex("E");
myGraph.addVertex("F");
myGraph.addVertex("G");
myGraph.addVertex("H");
myGraph.addWeightedEdge("A", "B", 5);
myGraph.addWeightedEdge("B", "D", 6);
myGraph.addWeightedEdge("D", "E", 3);
myGraph.addWeightedEdge("C", "D", 2);
myGraph.addWeightedEdge("A", "F", 1);
myGraph.addWeightedEdge("F", "G", 1);
myGraph.addWeightedEdge("G", "C", 1);
myGraph.addWeightedEdge("E", "H", 4);
myGraph.addWeightedEdge("C", "H", 8);
myGraph.display();
///////// Test 3
myGraph.FindDistance("A");
///////// Test 4
myGraph.FindShortestPath("A", "H");
myGraph.FindShortestPath("A", "B");
myGraph.FindShortestPath("C", "E");
myGraph.FindShortestPath("F", "D");
myGraph.FindShortestPath("C", "H");
```

The expected output is printed below:



```
42,G
   │      ├──48,E
   │      │      ├──53,F
   │      │      │      ├──
   │      │      │      └──
   │      │      └──66,B
   │      │      │      ├──
   │      │      │      └──
   │      └──63,C
   │      │      ├──71,A
   │      │      │      ├──
   │      │      │      └──
   │      │      └──65,I
   │      │      │      ├──74,D
   │      │      │      │      ├──
   │      │      │      │      └──
   │      │      │      └──79,H
   │      │      │      │      ├──
   │      │      │      │      └──
   │      │      │      │

G  E  F  C  I  B  A  D  H
Adjacency Matrix:

      A      B      C      D      E      F      G      H
A     -1     5      -1     -1     -1     1      -1     -1
B     -1     -1     -1     6      -1     -1     -1     -1
C     -1     -1     -1     2      -1     -1     -1     8
D     -1     -1     -1     -1     3      -1     -1     -1
E     -1     -1     -1     -1     -1     -1     -1     4
F     -1     -1     -1     -1     -1     -1     1      -1
G     -1     -1     1      -1     -1     -1     -1     -1
H     -1     -1     -1     -1     -1     -1     -1     -1

Distance from vertex A to vertex
A: 0
B: 5
C: 3
D: 5
E: 8
F: 1
G: 2
H: 11
Shortest Path from A to H: A-F-G-C-H
Shortest Path from A to B: A-B
Shortest Path from C to E: C-D-E
Shortest Path from F to D: F-G-C-D
Shortest Path from C to H: C-H
```