Critical Path Method -3rd hand in – Graphs

(By Emmely Lundberg cph-el69, classes on GitHub)

Assignment description:
The figure of a project graph below is from Wikipedia "Critical path method" (See last page). (One of the number is the diagram is not correct – find it).

Write a program with your own algorithms to find the values of the project graph. The algorithms need not to be fancy or optimal.

Make a representation of the tasks with connections and durations.
Calculate the earliest start and finish for each task.
Calculate the latest start and finish for each task.
Find the critical path.
Calculate the total floats.

Note: I got a hint to use two different adjacency list one for forward and one for backward direction so I made a quick change to the program before the handin.

"One of the number is the diagram is not correct – find it"

By looking at the graph and in my calculated project graph I have come to the conclusion that earliest finish for task C should be 35 and not 36.

"Write a program with your own algorithms to find the values of the project graph. The algorithms need not to be fancy or optimal."

I have written a program in java with my own implemented algorithm based on an example from geekforgeek. The program find all the possible project paths and calculate and prints the value for each path and task.The algorithm is not fancy or optimal.

The base for the algorithm has been taken from Print all paths from a given source to a destination the base of the code is contributed by Himanshu Shekhar. The original code represents the nodes as integer so I have modified that to use objects representing the tasks. I have also added an additional adjacency list so the graph can be traversed in backwards direction. The code is also changed so it can make the calculations on the tasks and print the results as JSON.

From Himanshu Shekhar article about the original algorithm:
"The idea is to do Depth First Traversal of given directed graph. Start the traversal from source. Keep storing the visited vertices in an array say 'path[]'. If we reach the destination

vertex, print contents of path[]. The important thing is to mark current vertices in path[] as visited also, so that the traversal doesn’t go in a cycle.”

“Make a representation of the tasks with connections and durations.”

- The tasks are represented as objects of the Vertex class. (See code Vertex class)
- The project graph is implemented as with two adjacency list one for forward direction that calculates earliest start/finish and one for backwards direction calculation the latest start/finish. The two adjacency list was used so the graph could be traversed backwards and forwards by a given source and destination node (task represented as Vertex object) keeping the right flow direction. (See code Adjacency List and Flow Control below)
- Makes calculations for each task per path in the graph. (see code Find all paths)
- Prints the paths as JSON. (A helper class results left out in this results)

“Find the critical path.”

To find the critical path I just select the path with the longest total duration.The program is not yet smart enough to take into consideration two paths with equally longest duration. *allPathsForward* in the Results class contains a list of all the paths.

Result:

```
------------------------------ Critical Path -------------------------------------

[{"A":{}},{"B":{}},{"C":{}},{"D":{}},{"E":{}},{"Total Duration":65}]
```

Logic:

```
  public void findCriticalPath(String title) {
      int max = 0;
      List<Vertex> criticalPath = null;

      for (List<Vertex> path : allPathsForward) {

          int sum = 0;
          for (Vertex i : path) {
              sum += i.duration;
          }
          if (sum > max) {
              max = sum;
              criticalPath = path;
          }
```

```
        }
        printHeadLine(title);
        PrintPathDynamic(criticalPath,new boolean[]{false,false,false,true});
        printPrintHeaderEnd();
    }
```

“Calculate the earliest start and finish for each task.

Result:

See below.

Logic:

The gaph is traversed forwards from source A to destination E, the *adjListForward* adjacency list is used.

```
for (Vertex i : adjListForward[u.id]) {

    if (!isVisited[i.id]) {
        // store current node
        // in path[]
        localPathList.add(i);
        findAllPathsLeftToRightUtil(i, d, isVisited, localPathList, results);
        // remove current node
        // in path[]
        localPathList.remove(i);
    }
}
```

When a path is found it traverse the nodes on that path and updates the *earlieststart/earliestfinish* of a node by the values from the previous node in the path.

The logic for when a path from source to destination is found looks like this:

```
if (u.equals(d)) {
    List<Vertex> finalPathFound = new ArrayList<Vertex>();
    Vertex previous = null;
        for (Vertex i : localPathList) {
            if (previous == null) {
                previous = i;
                previous.earliestStart = this.source.earliestStart;
                previous.earliestFinish = this.source.earliestFinish;
            }
            else {
                if (i.earliestStart == null || i.earliestStart <= previous.earliestFinish) {
                    i.earliestStart = previous.earliestFinish + 1;
                    i.earliestFinish = i.earliestStart + (i.duration - 1);
                }
            }
            previous = i;
            finalPathFound.add(i);
        }
        results.addPath(finalPathFound);

    }
```

Calculate the latest start and finish for each task. "

Logic:

The gaph is traversed backwards from source E to destination A, the backwards adjacency list, *adjListBackward,* is used.

When a path is found it traverse the nodes on that path and updates the *lateststart/latestfinish* of a node by the values from the previous node in the path.

The logic for when a path from source to destination is found looks like this:

```
    if (u.equals(d)) {

        List<Vertex> finalPathFound = new ArrayList<Vertex>();
        Vertex previous = null;

        for (Vertex i : localPathList) {
            if (previous == null) {
                previous = i;
                previous.latestStart = source.latestStart;//46;
                previous.latestFinish = source.earliestFinish;//65;
            }

            //System.out.print(i.name+" ");
            else {
                if (i.latestFinish == null || i.latestFinish < 0 || i.latestFinish >=
previous.latestStart) {
                    i.latestFinish  = previous.latestStart - 1;
                    i.latestStart = i.latestFinish - (i.duration - 1);
                }
            }
            previous = i;
            finalPathFound.add(i);
        }
        results.addPathBackwards(finalPathFound);

    }
```

Calculate the total floats.

Logic:

The floats are calculated like this where *allPathsForward* in the Results class contains a List of all the paths.

```
public void updateFloat() {
    for (List<Vertex> path : allPathsForward) {
        for (Vertex i : path) {
            i.floatTime = i.latestFinish - i.latestStart + 1;
        }
    }
}
```

Results:

```
------------------------------ Complete Project Graph -------------------------------------

[
        [{
                "A": {
                        "EarliestStart": 1,
                        "EarliestFinish": 10,
                        "LatestStart": 1,
                        "LatestFinish": 10,
                        "Float Time": 10
                }
        }, {
                "B": {
                        "EarliestStart": 11,
                        "EarliestFinish": 30,
                        "LatestStart": 11,
                        "LatestFinish": 30,
                        "Float Time": 20
                }
        }, {
                "C": {
                        "EarliestStart": 31,
                        "EarliestFinish": 35,
                        "LatestStart": 31,
                        "LatestFinish": 35,
                        "Float Time": 5
                }
        }, {
                "D": {
                        "EarliestStart": 36,
                        "EarliestFinish": 45,
                        "LatestStart": 36,
                        "LatestFinish": 45,
                        "Float Time": 10
                }
        }, {
                "E": {
                        "EarliestStart": 46,
                        "EarliestFinish": 65,
                        "LatestStart": 46,
                        "LatestFinish": 65,
                        "Float Time": 20
                }
        }, {
```

```
    "Total Duration": 65
}],
[{
    "A": {
        "EarliestStart": 1,
        "EarliestFinish": 10,
        "LatestStart": 1,
        "LatestFinish": 10,
        "Float Time": 10
    }
}, {
    "B": {
        "EarliestStart": 11,
        "EarliestFinish": 30,
        "LatestStart": 11,
        "LatestFinish": 30,
        "Float Time": 20
    }
}, {
    "C": {
        "EarliestStart": 31,
        "EarliestFinish": 35,
        "LatestStart": 31,
        "LatestFinish": 35,
        "Float Time": 5
    }
}, {
    "G": {
        "EarliestStart": 36,
        "EarliestFinish": 40,
        "LatestStart": 41,
        "LatestFinish": 45,
        "Float Time": 5
    }
}, {
    "E": {
        "EarliestStart": 46,
        "EarliestFinish": 65,
        "LatestStart": 46,
        "LatestFinish": 65,
        "Float Time": 20
    }
}, {
    "Total Duration": 60
}],
[{
        "A": {
                "EarliestStart": 1,
                "EarliestFinish": 10,
                "LatestStart": 1,
                "LatestFinish": 10,
                "Float Time": 10
        }
}, {
        "F": {
                "EarliestStart": 11,
                "EarliestFinish": 25,
                "LatestStart": 26,
                "LatestFinish": 40,
                "Float Time": 15
        }
}, {
        "G": {
                "EarliestStart": 36,
                "EarliestFinish": 40,
                "LatestStart": 41,
                "LatestFinish": 45,
                "Float Time": 5
        }
}, {
        "E": {
                "EarliestStart": 46,
                "EarliestFinish": 65,
                "LatestStart": 46,
                "LatestFinish": 65,
                "Float Time": 20
        }
}, {
        "Total Duration": 50
}],
[{
        "A": {
                "EarliestStart": 1,
                "EarliestFinish": 10,
                "LatestStart": 1,
                "LatestFinish": 10,
                "Float Time": 10
        }
}, {
        "H": {
                "EarliestStart": 11,
                "EarliestFinish": 25,
                "LatestStart": 31,
                "LatestFinish": 45,
                "Float Time": 15
            }
        }, {
            "E": {
                "EarliestStart": 46,
                "EarliestFinish": 65,
                "LatestStart": 46,
                "LatestFinish": 65,
                "Float Time": 20
            }
        }, {
            "Total Duration": 45
        }]
]
```

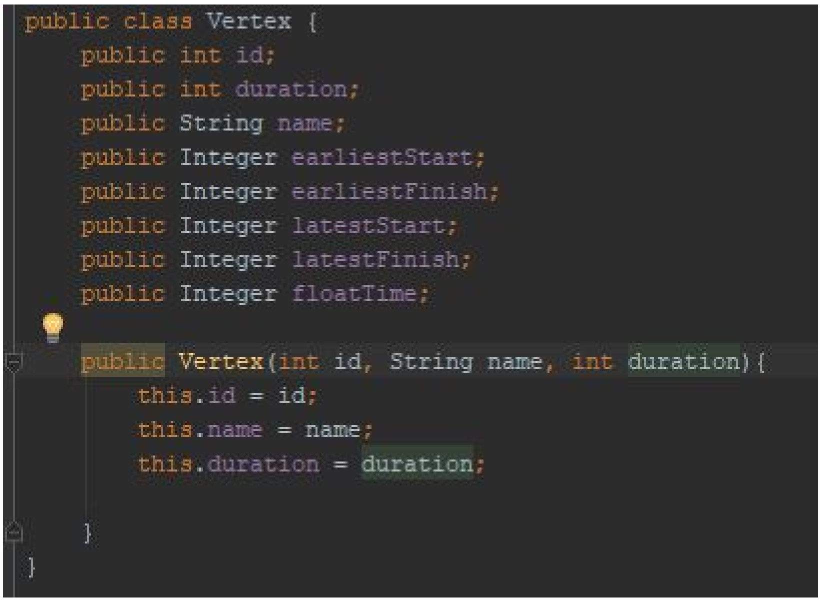
More on the code

The project tasks are represented as **objects called Vertex** and the project graph is represented in two **adjacency lists one** one for forward direction and one for backward direction in the Graph class. The graph is traversed in a depth first traversal to find all possible paths. When a path is found the program makes calculations such as earliest start/finish. For every calculation the Vertex object is updated and the paths are temporarily stored and printed with a helper method called Results.

I have introduced two adjacency list in the Graph class. This is because I made the graph traversable forward and backwards If I would have made the graph unidirectional the path A > F > G > C > D > E would be possible and this would have been unwanted behaviour.

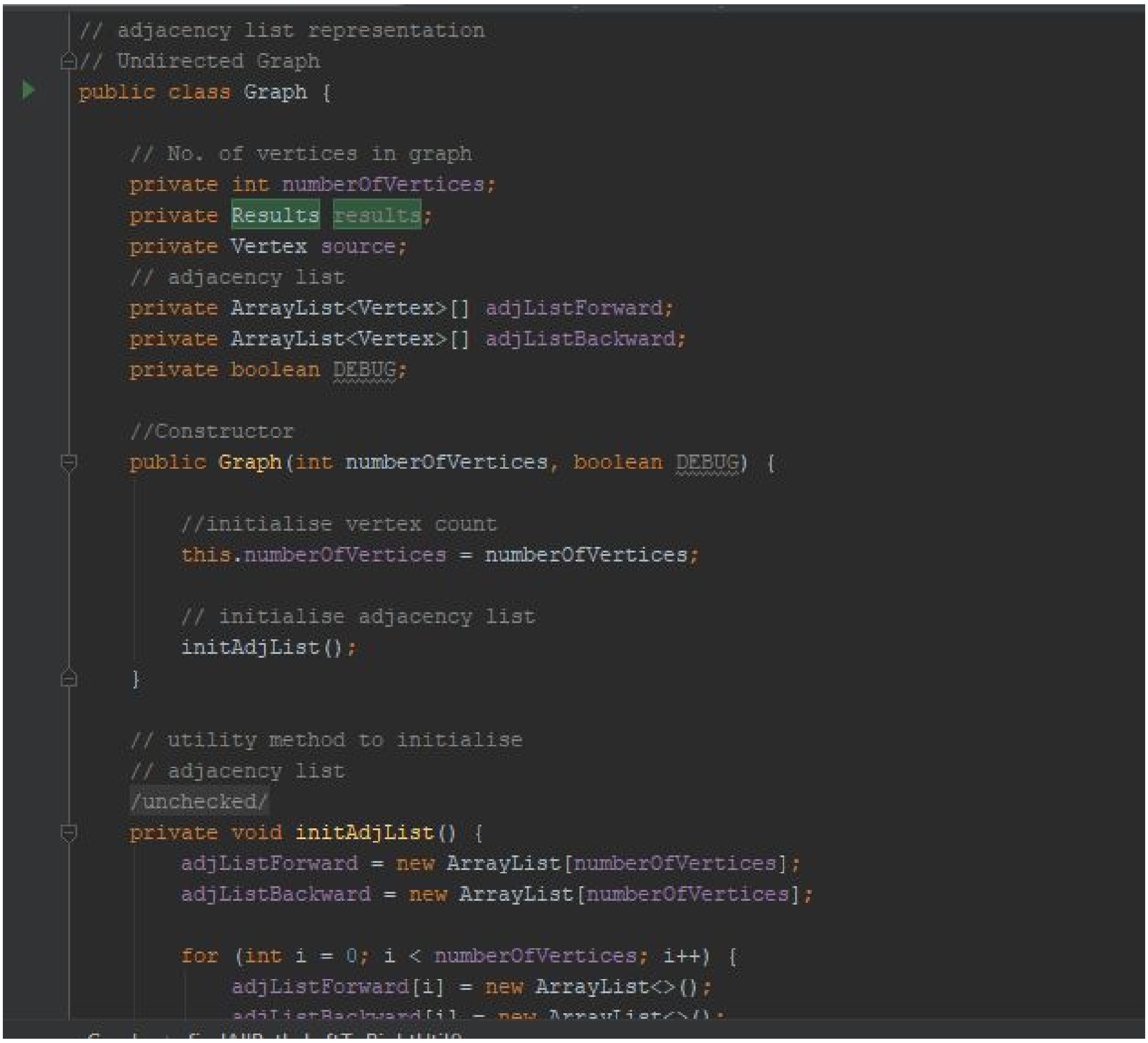
- Vertex class
- Adjacency
- Find all paths recursively
- Flow control
- Main class / Driver

Vertex class



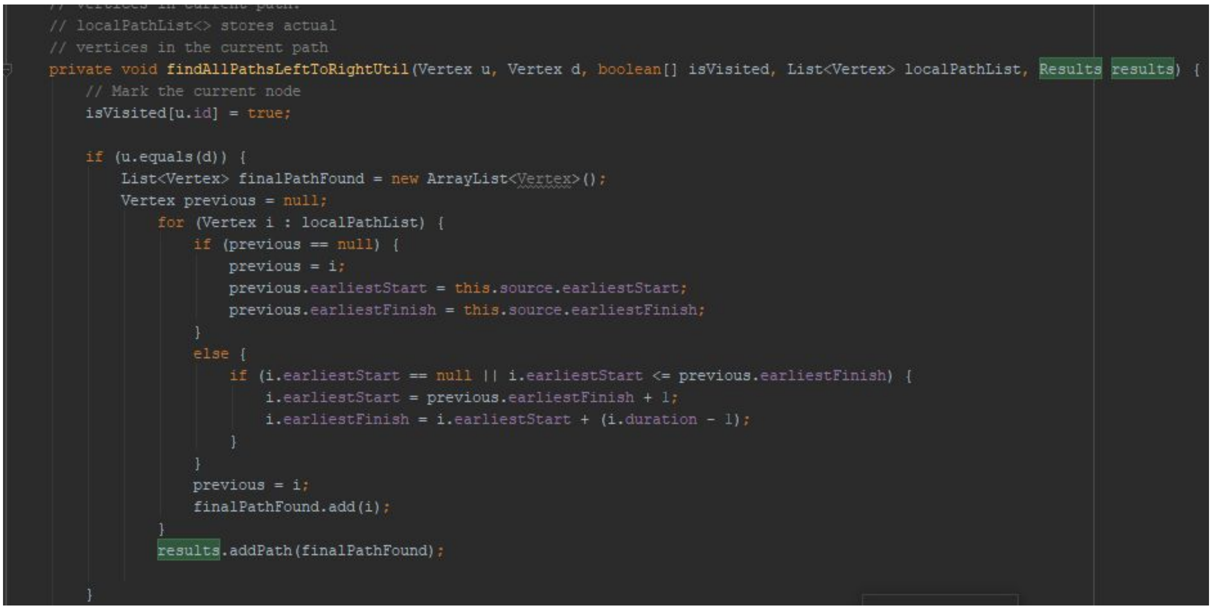
```java
public class Vertex {
    public int id;
    public int duration;
    public String name;
    public Integer earliestStart;
    public Integer earliestFinish;
    public Integer latestStart;
    public Integer latestFinish;
    public Integer floatTime;

    public Vertex(int id, String name, int duration){
        this.id = id;
        this.name = name;
        this.duration = duration;

    }
}
```

Adjacency List

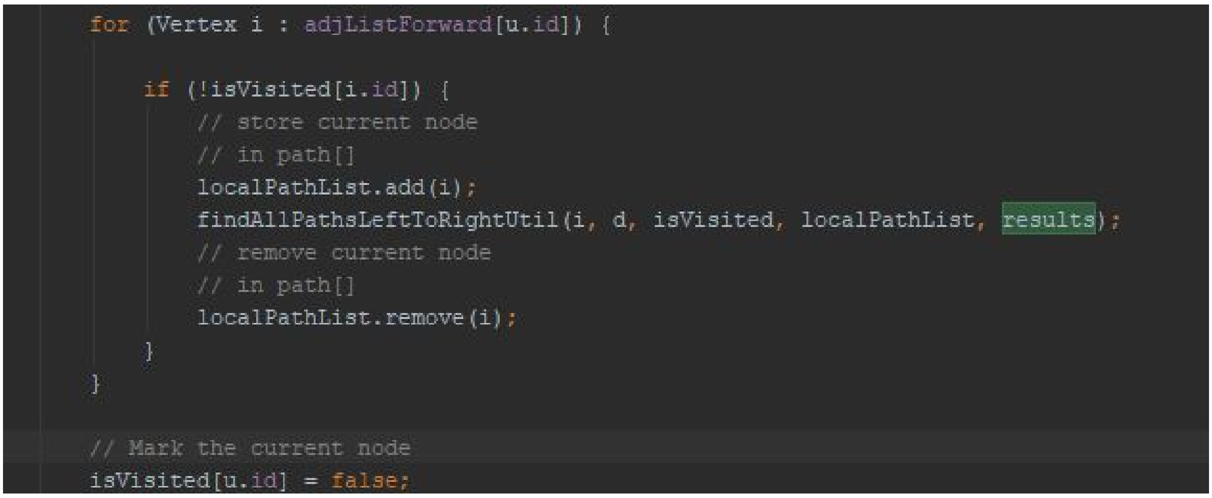


```
// adjacency list representation
// Undirected Graph
public class Graph {

    // No. of vertices in graph
    private int numberOfVertices;
    private Results results;
    private Vertex source;
    // adjacency list
    private ArrayList<Vertex>[] adjListForward;
    private ArrayList<Vertex>[] adjListBackward;
    private boolean DEBUG;

    //Constructor
    public Graph(int numberOfVertices, boolean DEBUG) {

        //initialise vertex count
        this.numberOfVertices = numberOfVertices;

        // initialise adjacency list
        initAdjList();
    }

    // utility method to initialise
    // adjacency list
    /unchecked/
    private void initAdjList() {
        adjListForward = new ArrayList[numberOfVertices];
        adjListBackward = new ArrayList[numberOfVertices];

        for (int i = 0; i < numberOfVertices; i++) {
            adjListForward[i] = new ArrayList<>();
            adjListBackward[i] = new ArrayList<>();
```

Find All Paths Recursively

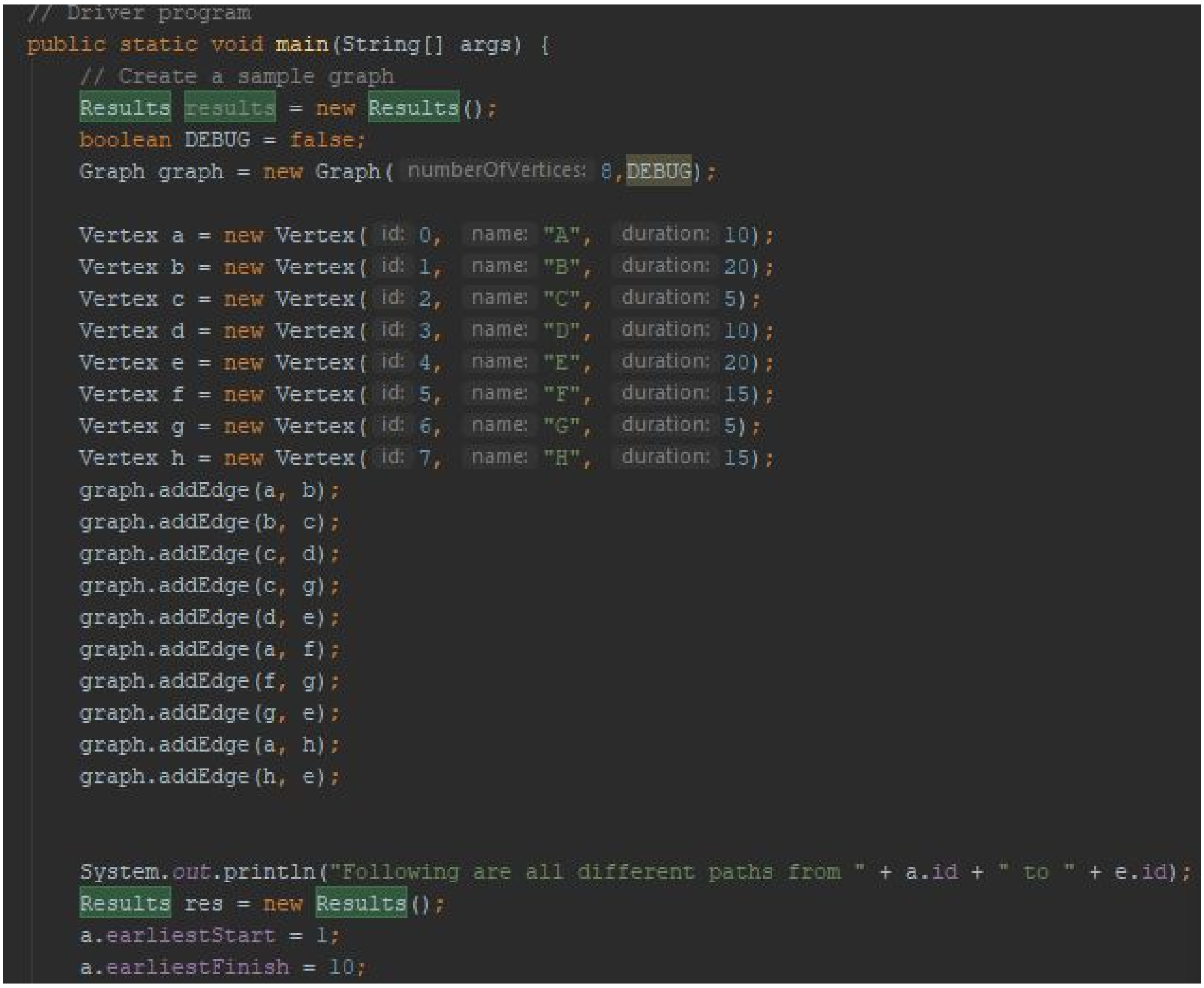


```java
    // vertices in current path.
    // localPathList<> stores actual
    // vertices in the current path
    private void findAllPathsLeftToRightUtil(Vertex u, Vertex d, boolean[] isVisited, List<Vertex> localPathList, Results results) {
        // Mark the current node
        isVisited[u.id] = true;

        if (u.equals(d)) {
            List<Vertex> finalPathFound = new ArrayList<Vertex>();
            Vertex previous = null;
                for (Vertex i : localPathList) {
                    if (previous == null) {
                        previous = i;
                        previous.earliestStart = this.source.earliestStart;
                        previous.earliestFinish = this.source.earliestFinish;
                    }
                    else {
                        if (i.earliestStart == null || i.earliestStart <= previous.earliestFinish) {
                            i.earliestStart = previous.earliestFinish + 1;
                            i.earliestFinish = i.earliestStart + (i.duration - 1);
                        }
                    }
                    previous = i;
                    finalPathFound.add(i);
                }
                results.addPath(finalPathFound);

        }
```

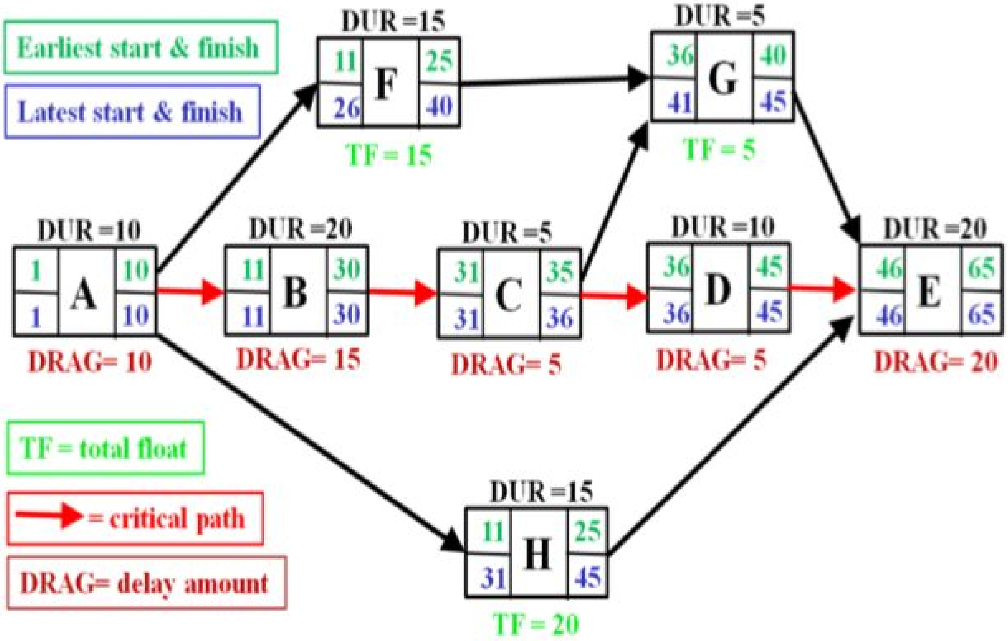


```java
        for (Vertex i : adjListForward[u.id]) {

            if (!isVisited[i.id]) {
                // store current node
                // in path[]
                localPathList.add(i);
                findAllPathsLeftToRightUtil(i, d, isVisited, localPathList, results);
                // remove current node
                // in path[]
                localPathList.remove(i);
            }
        }

        // Mark the current node
        isVisited[u.id] = false;
```

Main



```
// Driver program
public static void main(String[] args) {
    // Create a sample graph
    Results results = new Results();
    boolean DEBUG = false;
    Graph graph = new Graph( numberOfVertices: 8,DEBUG);

    Vertex a = new Vertex( id: 0,  name: "A",  duration: 10);
    Vertex b = new Vertex( id: 1,  name: "B",  duration: 20);
    Vertex c = new Vertex( id: 2,  name: "C",  duration: 5);
    Vertex d = new Vertex( id: 3,  name: "D",  duration: 10);
    Vertex e = new Vertex( id: 4,  name: "E",  duration: 20);
    Vertex f = new Vertex( id: 5,  name: "F",  duration: 15);
    Vertex g = new Vertex( id: 6,  name: "G",  duration: 5);
    Vertex h = new Vertex( id: 7,  name: "H",  duration: 15);
    graph.addEdge(a, b);
    graph.addEdge(b, c);
    graph.addEdge(c, d);
    graph.addEdge(c, g);
    graph.addEdge(d, e);
    graph.addEdge(a, f);
    graph.addEdge(f, g);
    graph.addEdge(g, e);
    graph.addEdge(a, h);
    graph.addEdge(h, e);


    System.out.println("Following are all different paths from " + a.id + " to " + e.id);
    Results res = new Results();
    a.earliestStart = 1;
    a.earliestFinish = 10;
```


Earliest start & finish
Latest start & finish
DUR =15
11
F
25
26
40
TF = 15
DUR =5
36
G
40
41
45
TF = 5
DUR =10
1
A
10
1
10
DRAG= 10
DUR =20
11
B
30
11
30
DRAG= 15
DUR =5
31
C
35
31
36
DRAG= 5
DUR =10
36
D
45
36
45
DRAG= 5
DUR =20
46
E
65
46
65
DRAG= 20
TF = total float
= critical path
DRAG= delay amount
DUR =15
11
H
25
31
45
TF = 20