PA2 Report

CSI2110 - Data Structures and Algorithms

Fall 2023

School of Electrical Engineering

&

Computer Science University of Ottawa

Professor: Lucia Moura

Email: lmoura@uottawa.ca

Hengjing Zhang - 300288003

Submission Date: 11/26/2023

Description for “Graph” Class

The Edge List approach directly stores each edge in the graph as a separate object. In the provided implementation, each Edge object encapsulates two vertices (v1 and v2) representing the connected stations and an associated weight, which denotes the travel time between these stations.

1. Two Inner Classes (Vertex and Edge)

- **Vertex Class:** Represents a station in the Paris Metro network.
- **Edge Class:** Each edge links two Vertex objects (v1 and v2) and has an associated weight representing the travel time between these stations.

2. Data Structures and Algorithms

- **Vertices Array:** The vertices array is used to store all the vertices (stations) in the graph.
- **Edges Array:** Holds all the edges in the graph. The array supports the storage of all connections in the network, including their travel times.
- **Static class Entry (Priority Queue):** It holds a label (used as a priority) and a reference to a “Vertex”. The priority queue is instrumental in selecting the next vertex to process based on the shortest cumulative weight.
- **LinkedList for Incident Edges:** Used in the *getIncidentEdges* method to store edges connected to a specific vertex. LinkedLists are chosen for their dynamic nature, allowing for easy addition and traversal of edges, which is particularly useful in methods like *getAllStations* and *getShortestPath*.

Algorithms used to answer the questions in task item 2

i) Identify all the stations belonging to the same line of a given station.

- **Overview**

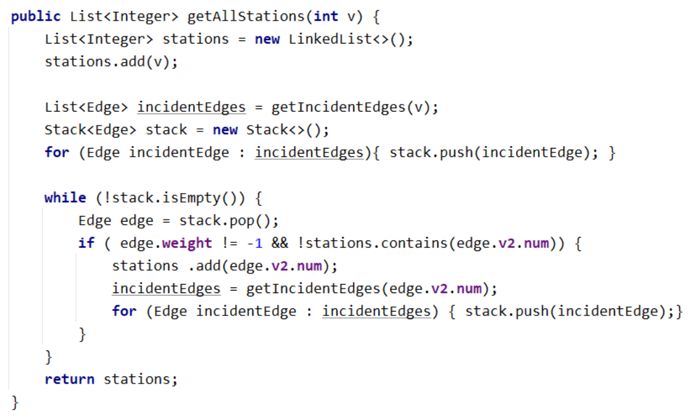
The method *getAllStations* was implemented to solve this question. The *getAllStations* method takes a station identifier (N1) as input and returns a list containing all stations that belong to the same line as the input station. The method starts by adding the input station to a list of stations, then retrieves all edges directly connected to this station using the *getIncidentEdges* method.

- **Implementation of Depth-First Search**

A stack is created to store the edges to be visited. Edges directly connected to the initial station are added to the stack.

While the stack is not empty, an edge is popped from the stack for processing. The method checks if the target vertex of the edge has already been visited (i.e., whether it is already in the list of stations). If not visited, and if the edge is not a walking connection (weight not equal to -1), the target vertex of the edge is added to the list of stations. All edges connected to this vertex are then added to the stack.

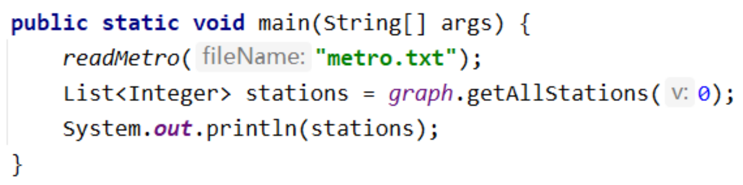
The traversal continues until the stack is empty. At this point, all stations belonging to the same line as the initial station have been added to the list.



```java
public List<Integer> getAllStations(int v) {
    List<Integer> stations = new LinkedList<>();
    stations.add(v);

    List<Edge> incidentEdges = getIncidentEdges(v);
    Stack<Edge> stack = new Stack<>();
    for (Edge incidentEdge : incidentEdges){ stack.push(incidentEdge); }

    while (!stack.isEmpty()) {
        Edge edge = stack.pop();
        if ( edge.weight != -1 && !stations.contains(edge.v2.num)) {
            stations .add(edge.v2.num);
            incidentEdges = getIncidentEdges(edge.v2.num);
            for (Edge incidentEdge : incidentEdges) { stack.push(incidentEdge);}
        }
    }
    return stations;
}
```

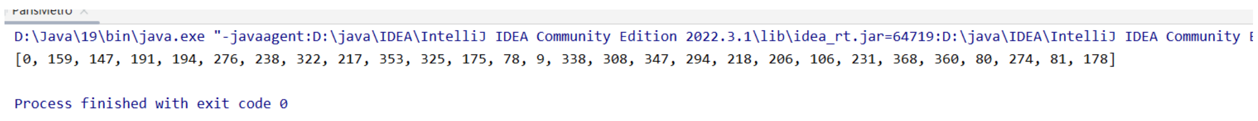
Figure1: Code for method *getAllStations*

- **Results**



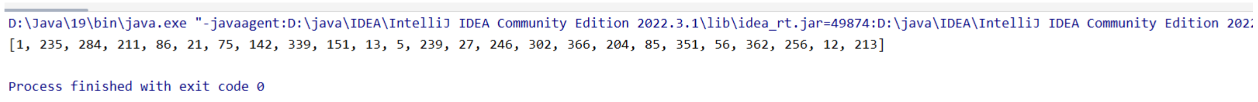
```java
public static void main(String[] args) {
    readMetro( fileName: "metro.txt");
    List<Integer> stations = graph.getAllStations( v: 0);
    System.out.println(stations);
}
```

Figure2: Command line in main method for class ParisMetro



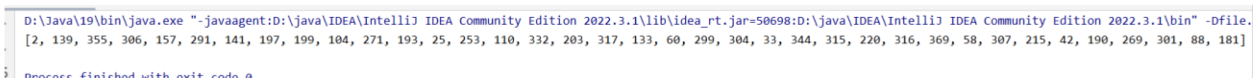
```
D:\Java\19\bin\java.exe "-javaagent:D:\java\IDEA\IntelliJ IDEA Community Edition 2022.3.1\lib\idea_rt.jar=64719:D:\java\IDEA\IntelliJ IDEA Community E
[0, 159, 147, 191, 194, 276, 238, 322, 217, 353, 325, 175, 78, 9, 338, 308, 347, 294, 218, 206, 106, 231, 368, 360, 80, 274, 81, 178]

Process finished with exit code 0
```

Result 1: N1 = 0



```
D:\Java\19\bin\java.exe "-javaagent:D:\java\IDEA\IntelliJ IDEA Community Edition 2022.3.1\lib\idea_rt.jar=49874:D:\java\IDEA\IntelliJ IDEA Community Edition 202
[1, 235, 284, 211, 86, 21, 75, 142, 339, 151, 13, 5, 239, 27, 246, 302, 366, 204, 85, 351, 56, 362, 256, 12, 213]

Process finished with exit code 0
```

Result 2: N1 = 1



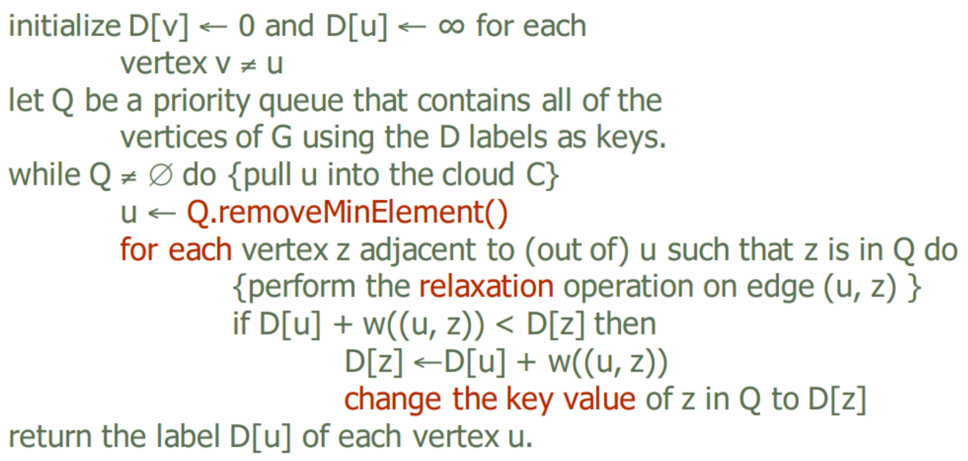
```
D:\Java\19\bin\java.exe "-javaagent:D:\java\IDEA\IntelliJ IDEA Community Edition 2022.3.1\lib\idea_rt.jar=50698:D:\java\IDEA\IntelliJ IDEA Community Edition 2022.3.1\bin" -Dfile.
[2, 139, 355, 306, 157, 291, 141, 197, 199, 104, 271, 193, 25, 253, 110, 332, 203, 317, 133, 60, 299, 304, 33, 344, 315, 220, 316, 369, 58, 307, 215, 42, 190, 269, 301, 88, 181]

Process finished with exit code 0
```

Result 3: N1 = 2

ii) Find the shortest path between any two stations.

- **Overview**

Method *getShortestPath* is designed to find the shortest path between two stations, identified by v1 and v2, in terms of total travel time. A priority queue (PriorityQueue<Entry>) is initialized to manage vertices during the search process. The Entry class holds a vertex and its corresponding label (distance from the start vertex). Additionally, two arrays, *D* and *array*, are initialized. *D* stores the shortest distance from the start vertex (v1) to every other vertex. *array* represents the next point.



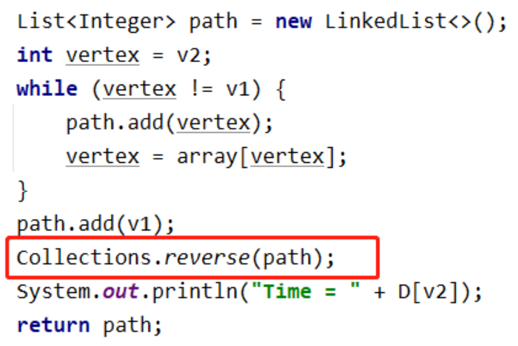
```
initialize D[v] ← 0 and D[u] ← ∞ for each
        vertex v ≠ u
let Q be a priority queue that contains all of the
        vertices of G using the D labels as keys.
while Q ≠ ∅ do {pull u into the cloud C}
        u ← Q.removeMinElement()
        for each vertex z adjacent to (out of) u such that z is in Q do
                {perform the relaxation operation on edge (u, z) }
                if D[u] + w((u, z)) < D[z] then
                        D[z] ←D[u] + w((u, z))
                        change the key value of z in Q to D[z]
return the label D[u] of each vertex u.
```

Figure3: Pseudocode for Dijkstra's Algorithm

- **Algorithm Implementation**

The method iteratively extracts the vertex with the minimum distance label from the priority queue. For each extracted vertex (**u**), the method updates the distance to its adjacent vertices (**z**) if a shorter path is found. This is where the method checks if an edge is a walking connection (weight = -1) and assigns it a fixed travel time of 90 seconds. The predecessor of each vertex in the shortest path is updated in the array.

Once the destination vertex (v2) is reached, the method reconstructs the shortest path by backtracking from the destination to the start vertex using the array that stored predecessors. The path is then reversed to present it from start to destination.



```
List<Integer> path = new LinkedList<>();
int vertex = v2;
while (vertex != v1) {
    path.add(vertex);
    vertex = array[vertex];
}
path.add(v1);
Collections.reverse(path);
System.out.println("Time = " + D[v2]);
return path;
```

Figure 4: reverse the path to get the shortest path.

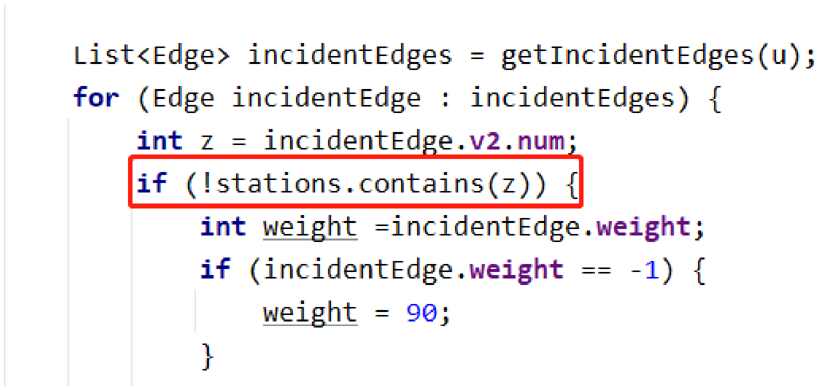
iii) Find the shortest path between two stations when we have the information that one given line is not functioning.

- **Overview**

  Similar to the previous method, a priority queue (PriorityQueue<Entry>) is used for managing the vertices. Two arrays, *D* and *array*, are initialized. However, the initialization differs slightly in that vertices corresponding to the same line as v3 are not added to the priority queue.

- **Modified Dijkstra's Algorithm**

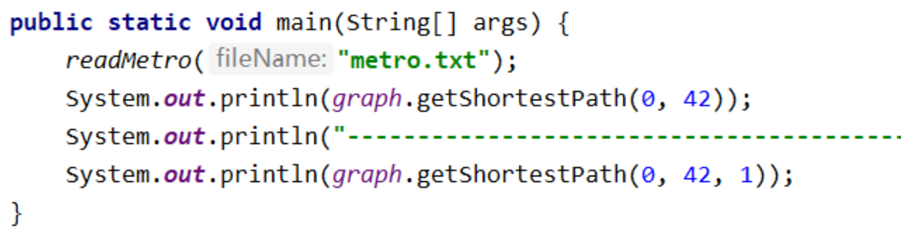
  The method employs a variation of Dijkstra's algorithm. It iteratively processes vertices with the minimum distance, updating the distances of adjacent vertices. A crucial difference here is the exclusion of vertices that are on the same line as v3. If an adjacent vertex (z) is on the same line as v3, it is not considered for updating the distance. As in the standard method, if an edge represents a walking connection (weight = -1), a fixed time of 90 seconds is used. The part of the code is given below.



```
    List<Edge> incidentEdges = getIncidentEdges(u);
    for (Edge incidentEdge : incidentEdges) {
        int z = incidentEdge.v2.num;
        if (!stations.contains(z)) {
            int weight =incidentEdge.weight;
            if (incidentEdge.weight == -1) {
                weight = 90;
            }
```

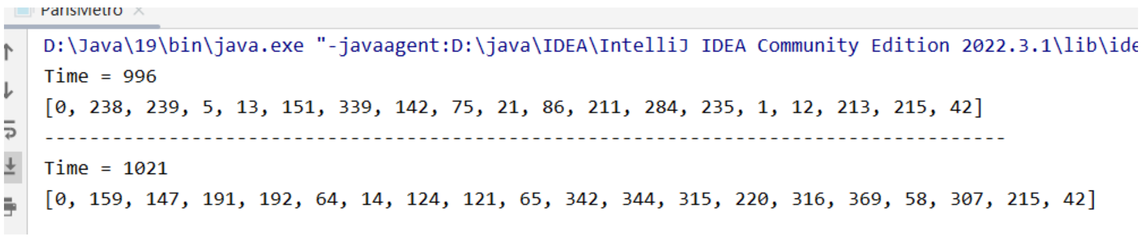
Figure 5: Modified Dijkstra's Algorithm

Results for ii) and iii) together



```
public static void main(String[] args) {
    readMetro( fileName: "metro.txt");
    System.out.println(graph.getShortestPath(0, 42));
    System.out.println("---------------------------------------
    System.out.println(graph.getShortestPath(0, 42, 1));
}
```

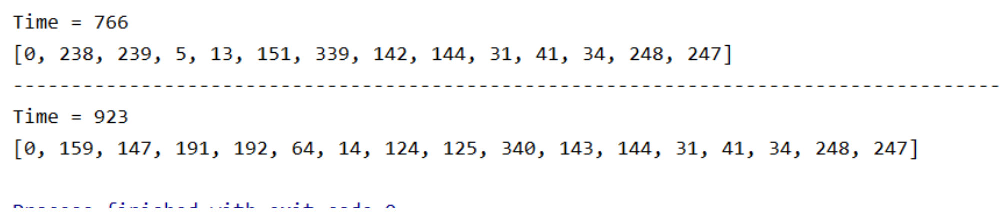
Figure 6: Command line in main method



```
D:\Java\19\bin\java.exe "-javaagent:D:\java\IDEA\IntelliJ IDEA Community Edition 2022.3.1\lib\ide
Time = 996
[0, 238, 239, 5, 13, 151, 339, 142, 75, 21, 86, 211, 284, 235, 1, 12, 213, 215, 42]
---------------------------------------------------------------------------------
Time = 1021
[0, 159, 147, 191, 192, 64, 14, 124, 121, 65, 342, 344, 315, 220, 316, 369, 58, 307, 215, 42]
```

Result 4: N1 = 0, N2 = 42, Time = 996

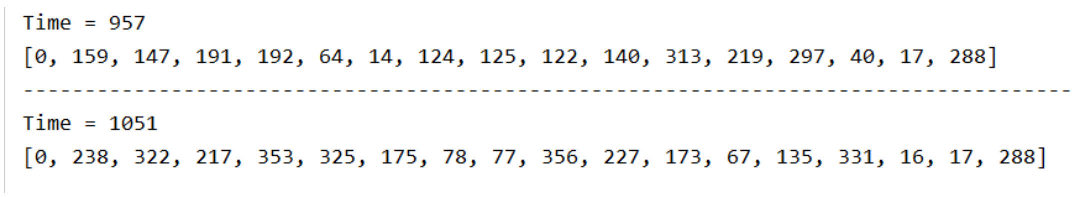
N1 = 0, N2 = 42, N3 = 1, Time = 1021\



```
Time = 766
[0, 238, 239, 5, 13, 151, 339, 142, 144, 31, 41, 34, 248, 247]
---------------------------------------------------------------------------------
Time = 923
[0, 159, 147, 191, 192, 64, 14, 124, 125, 340, 143, 144, 31, 41, 34, 248, 247]
```

Result 5: N1 = 0, N2 = 247, Time = 766

N1 = 0, N2 = 247, N3 = 31, Time = 923



```
Time = 957
[0, 159, 147, 191, 192, 64, 14, 124, 125, 122, 140, 313, 219, 297, 40, 17, 288]
---------------------------------------------------------------------------------
Time = 1051
[0, 238, 322, 217, 353, 325, 175, 78, 77, 356, 227, 173, 67, 135, 331, 16, 17, 288]
```

Result 6: N1 = 0, N2 = 288, Time = 957

N1 = 0, N2 = 288, N3 = 3, Time = 1051

References

Lecture 17 - SPT112 page 14