Assignment #1

Algorithm and Analysis COSC 2123/1285

Submission Deadline: 11:59pm Tuesday, 16th of April, 2019

Members/Student ID: Syed Hariz/S3701799, Simon Hesjevik/S3694451

Table of Contents

Data Structure

Adjacent List

The data structure used to store vertices of an adjacent list are; HashMap, an array and a custom LinkedList to efficiently search, add and delete a data. The logic that was applied on implementing this class is having a HashMap that accepts a parameter type <String, Integer>, an array of Nodes and a LinkedList of the vertex's pairs.

HashMap<String, Integer>

- The functionality of applying the HashMap data structure into this class is to store a set of different nodes and its' vertex label that was created in the Adjacent List graph where the key of the HashMap parameter String is the vertex label and parameter Integer is the location of the node in the array. Illustration of a node in the graph can be seen as shown in Figure 1.

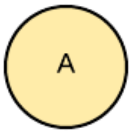


Figure 1 – A node in Adjacent List with vertex label 'A'

- How HashMap is invoked is when the graph adds a new node with its vertex label. In the AdjList class when the addVertex(String vertLabel) is called, the function accepts the label of the vertex as its parameter which will be used as the key of a node in Hashmap<String, Integer>. The class then processes the algorithm to store the exact location in a Node array where this value will be passed in the 2nd parameter of the HashMap, Integer.

Node[] array

- The Node[] array is used to store a node Object. In each node Object, contains a custom Linked List Object that stores the pairs of a node in the graph. The relationship between a node and its' pairs is established whenever an edge is added between the 2 different nodes. In Figure 3, is an example of how Vertex 'A' has a relationship with Vertex 'B'. Figure 4 is a class diagram of the Node class.

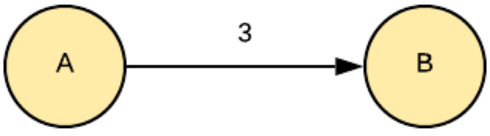


Figure 3 – A relationship between 2 vertices with edge weight 3

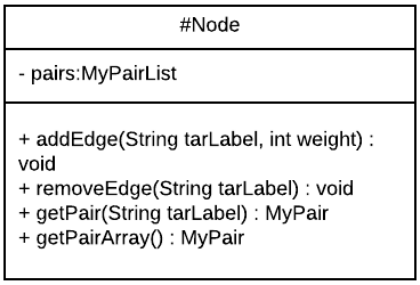


Figure 4 – Node Class Activity Diagram

MyPairList

- MyPairList is a custom array that is designed to fulfil a set of functionalities of a LinkedList array. This class will store pairs of a vertex that was added into the graph by creating an Object of MyPair class that stores the value of TargetLabel and its EdgeWeight. A class diagram of this class is shown in Figure 5.

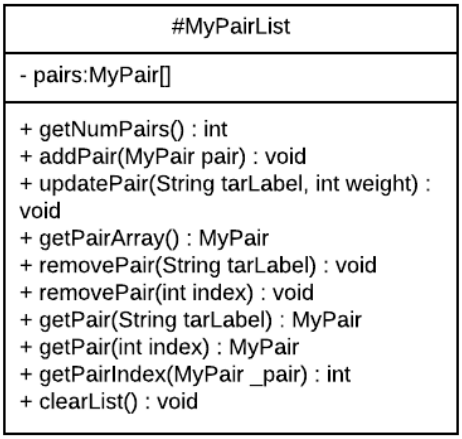


Figure 5 – MyPairList Class Activity Diagram

Incidence Matrix

The data structure used to store vertices of an incidence matrix are; HashMap, a 2D array to create the incidence matrix graph and 2 different types array; String and MyPair.

HashMap<String, Integer>

- Same as the HashMap functionality used in Adjacent List. This HashMap also accepts parameter type of <String, Integer>. The String parameter acts as the key which stores the label of vertex, the Integer parameter is used to position the vertex in the 2D array column.

2D array

- The 2D array is to create the incidence matrix graph. Which columns and rows of the array indicates the label of a vertex and the size of the array depends on how many vertices are created in the graph.

String array

- A String array is used to store all the vertices that were created on the graph. This array basically holds the vertex label of a node.

MyPairs array

- MyPairs array is used to store the pairs of a vertex when an edge is added between 2 vertices.

Data generation

Adjacent List

Operations

- Adding vertex

  This observation is made to time the operation of adding vertices into the graph and how it varies when more vertices are added into the graph. System.nanoTime() is used to clock the following operation. It will then be converted into Seconds for better readability.

  Code to measure time;

```
long startTime = System.nanoTime();
list.addVertex("A");
long totalTime = System.nanoTime() - startTime;
double elapsedTimeInSecond = (double) totalTime / 1_000_000_000;
System.out.println( "Total time: " + elapsedTimeInSecond
      + " seconds" );
```

  Adding 1 vertex:

```
Total time: 0.001955271 seconds
```

  Adding 2 vertices:

```
Total time: 0.002037191 seconds
```

  Adding 3 vertices:

```
Total time: 0.002077013 seconds
```

We can conclude that when more vertices are added into the graph, the longer period of time it takes to create the graph. The number of nodes that were created will affect the time of creating the graph as it will go call the class method multiple times.

Algorithm for adding a vertex:

CHECK if node array is bigger than the current node amount + 1

INCREASE array by 1 and copy over the items if the CHECK is true

INSTANTIATE a new node and set it to the newly created element in the node array

ADD the vertex label to the node HashMap and set the key to the index of the last element in the node array.

INCREMENT the numNodes by 1.

- Adding an edge
  This operation is to measure the performance of the graph when edges are added between 2 vertices. First vertices are added to the graph, then edges are added between 2 or more vertices and clock the time of running this operation.

  Code to measure time;

```
long startTime = System.nanoTime();
list.addVertex("A");
list.addVertex("B");
list.addEdge("A", "B", 1);
long totalTime = System.nanoTime() - startTime;
double elapsedTimeInSecond = (double) totalTime / 1_000_000_000;
System.out.println( "Total time: " + elapsedTimeInSecond
            + " seconds" );
```

  Adding 2 vertices, 1 edge:

```
Total time: 0.001972906 seconds
```

  Adding 3 vertices, 1 edge:

```
Total time: 0.00218624 seconds
```

  Adding 3 vertices, 2 edges:

```
Total time: 0.002500266 seconds
```

In this operation, it takes more time to add vertices and edges in the graph. By comparing the stats on adding vertices alone, it takes higher time when edges are being added as well. We can conclude that when more operations are added the longer time it takes for the graph to create.

Scenario 1: Shrinking graphs

- Removing Vertices
  This operation is made to time the operation of removing vertices from the graph. We can time the operation when removing different amounts of vertices to get an idea of how efficient it is. System.nanoTime() is again used to time the operation. It will then print out the elapsed time.

  Original state of graph: 8 vertices, 8 edges
  Graph Density: 0.125

  Test 1 - Removing 1 vertex:

```
Total Time: 2.5031E-5 seconds
```

  Graph state after removal: 7 vertices, 7 edges
  Graph Density: 0.142

Test 2 - Removing 2 vertices:

```
Total Time: 3.2995E-5 seconds
```

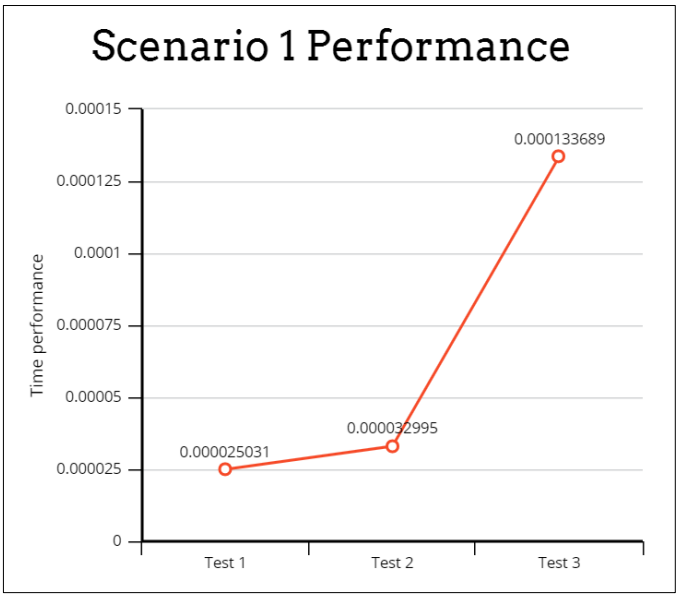
Graph state after removal: 6 vertices, 5 edges
Graph Density: 0.138

Test 3 - Removing 4 vertices:

```
Total Time: 1.33689E-4 seconds
```

Graph state after removal: 4 vertices, 3 edges
Graph Density: 0.1875

The performance when deleting a vertex from the graph varies because when the collection of nodes in the data structure gets smaller, the easier it is to search for the correct node and remove them from the collection. Hence, the smaller the amount of vertex stored, the faster to remove.



- Removing Edges
  This operation is made to time the operation of removing edges from the graph. We can time the operation when removing different amounts of edges to get an idea of how efficient it is. System.nanoTime() is again used to time the operation. It will then print out the elapsed time.

  Original state of graph: 8 vertices, 8 edges
  Graph Density: 0.125

  Test 1 - Removing 1 edge:

  ```
  Total Time: 1.0809E-5 seconds
  ```

  Graph state after removal: 8 vertices, 7 edges
  Graph Density: 0.109

Test 2 - Removing 2 edges:

```
Total Time: 1.4791E-5 seconds
```

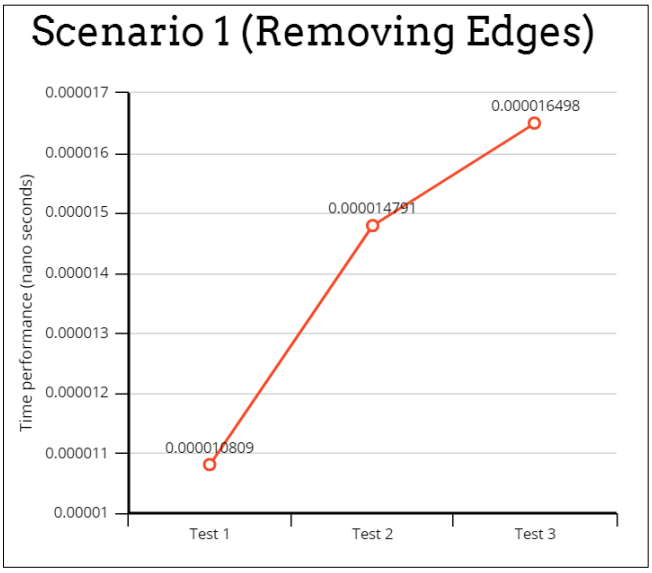
Graph state after removal: 8 vertices, 6 edges
Graph Density: 0.071

Test 3 - Removing 4 edges:

```
Total Time: 1.6498E-5 seconds
```

Graph state after removal: 8 vertices, 4 edges
Graph Density: 0.062

Based on observation, the longer time is needed when removing a higher amount of edges this is because the same function is being called to remove an edge. The function to remove an edge, has if-else statements which means that when it is called it has to pass every statement to find which arguments to run.



Scenario 2: Nearest neighbours

- In neighbours of a vertex
  This operation getting all the edges that go to a certain vertex. This takes a parameter that limits how many edges to list after it has been sorted.

  Original state of graph: 8 vertices, 8 edges
  Graph Density: 0.125

  Test 1 - In neighbours of 1 vertex, k value = -1:

  ```
  Total Time: 0.002566826 seconds
  ```

  Test 2 - In neighbours of 2 vertex, k value = -1:

  ```
  Total Time: 0.002746026 seconds
  ```

Test 3 - In neighbours of 2 vertex, k value = 2:

```
Total Time: 0.001711217 seconds
```

- Out neighbours of a vertex
  This operation getting all the edges that go from a certain vertex. This takes a parameter that limits how many edges to list after it has been sorted.

  Original state of graph: 8 vertices, 8 edges
  Graph Density: 0.125

  Test 1 - Out neighbours of 1 vertex, k value = -1:

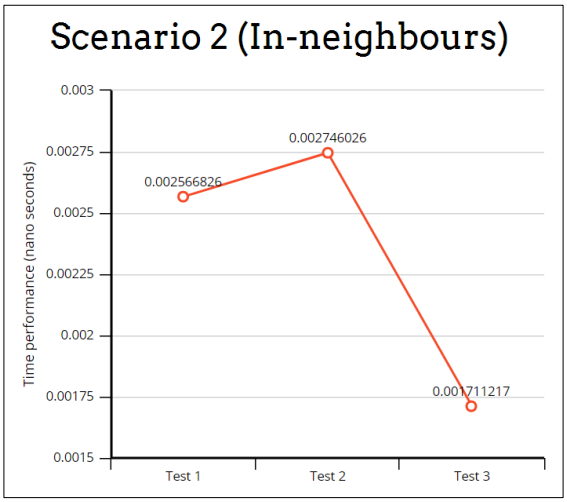
  ```
  Total Time: 0.002174293 seconds
  ```

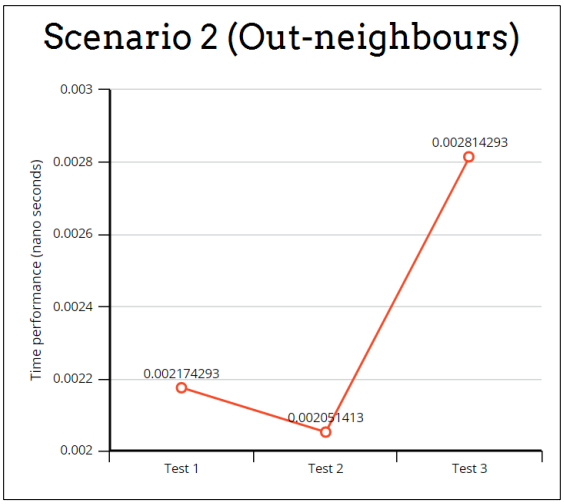
  Test 2 - Out neighbours of 2 vertex, k value = -1:

  ```
  Total Time: 0.002051413 seconds
  ```

  Test 3 - Out neighbours of 2 vertex, k value = 2:

  ```
  Total Time: 0.002814293 seconds
  ```





Scenario 3: Changing associations

- This operation changes the weight of an edge in the graph.

  Original state of graph: 8 vertices, 8 edges
  Graph Density: 0.125

  Test 1 - Change 1 edge weight from 7 to 5:

  ```
  Total Time: 7.396E-6 seconds
  ```

  Test 2 - Change edge weights of 2 different node relationship:

  ```
  Total Time: 3.2426E-5 seconds
  ```

Test 3 - Change edge weights of 4 different node relationship:

```
Total Time: 1.3084E-5 seconds
```

Graph state after removal: 7 vertices, 7 edges
Graph Density: 0.142

Test 1 - Change 1 edge weight from 7 to 5:

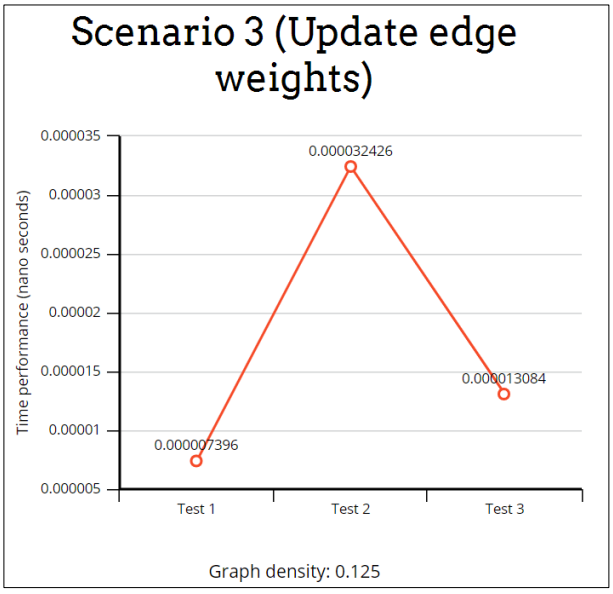
```
Total Time: 2.845E-6 seconds
```

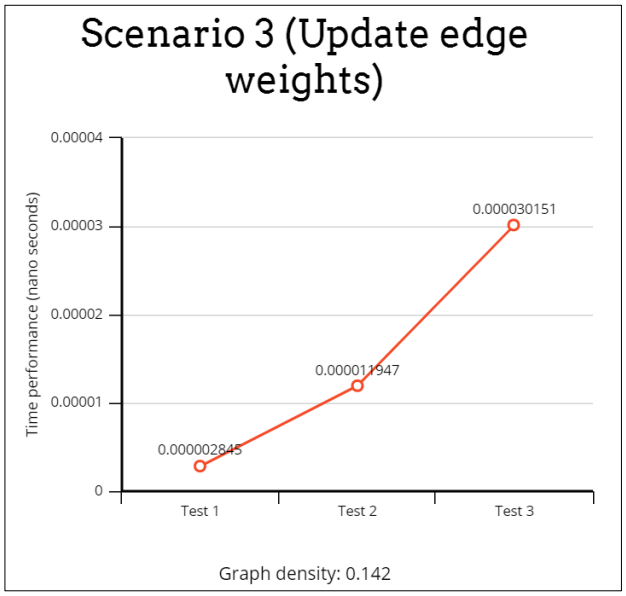
Test 2 - Change edge weights of 2 different node relationship:

```
Total Time: 1.1947E-5 seconds
```

Test 3 - Change edge weights of 4 different node relationship:

```
Total Time: 3.0151E-5 seconds
```





Incidence Matrix

Scenario 1: Shrinking graphs

- Removing Vertex
  This operation removed a vertex from the matrix. It takes in a parameter of which vertex label it should remove from the said matrix.

  Test 1 – Removing 1 vertex

  ```
  Total Time: 5.81465E-4 seconds
  ```

  Test 2 – Removing 2 vertices

  ```
  Total Time: 6.15668E-4 seconds
  ```

  Test 3 – Removing 3 vertices

  ```
  Total Time: 4.93663E-4 seconds
  ```

- Removing Edges
  This operation removes edges from the incidence matrix.

  Test 1 – Removing 1 edge

  ```
  Total Time: 2.46127E-4 seconds
  ```

  Test 2 – Removing 2 edges

  ```
  Total Time: 1.84419E-4 seconds
  ```

  Test 3 – Removing 3 edges

  ```
  Total Time: 2.41542E-4 seconds
  ```

Unlike the adjacent list, the remove methods in the matrix class don't depend on how many edges or vertices. This is because when it changes the edge it edits one location in the matrix instead of looping through a list of vertices.

Scenario 2: Nearest neighbours

- In-neighbours

  Test 1 – In 1 edge

  ```
  Total Time: 0.001313144 seconds
  ```

  Test 2 – In 2 edges

  ```
  Total Time: 0.00122499 seconds
  ```

  Test 3 – In 3 edge

  ```
  Total Time: 0.001162225 seconds
  ```

- Out-neighbours

  Test 1 – out 1 edge

  ```
  Total Time: 0.001162225 seconds
  ```

  Test 2 – out 2 edges

  ```
  Total Time: 0.001782352 seconds
  ```

  Test 3 – out 3 edges

  ```
  Total Time: 0.002195234 seconds
  ```

The results here show us that the getInNeighbours method depends on the number of vertices. This is because of our implementation stores all the edges in an array. The more edges exist the longer it takes to find the correct ones.

Scenario 3: Changing associations

- This operation changes the weight of an edge in the matrix.
  Original state of graph: 8 vertices, 8 edges
  Graph Density: 0.125
  Test 1 - Change 1 edge weight from 7 to 5:

  ```
  Total Time: 5.689E-6 seconds
  ```

  Test 2 - Change edge weights of 2 different node relationship:

  ```
  Total Time: 1.3084E-5 seconds
  ```

  Test 3 - Change edge weights of 4 different node relationship:

  ```
  Total Time: 1.7066E-5 seconds
  ```

  Graph state after removal: 7 vertices, 7 edges
  Graph Density: 0.142

  Test 1 - Change 1 edge weight from 7 to 5:

  ```
  Total Time: 4.551E-6 seconds
  ```

  Test 2 - Change edge weights of 2 different node relationship:

  ```
  Total Time: 6.258E-6 seconds
  ```

  Test 3 - Change edge weights of 4 different node relationship:

  ```
  Total Time: 6.258E-6 seconds
  ```

These results show us that the updateWeightEdge method doesn't change its timing depending on the number of edges. This is because the weight values are stored in a 2d array. There is no need to iterate through a list to change the values.

Analysis

In the adjacent list graph, my recommendation on improving the performance of this graph is by having a data structure of HashMap that accepts parameter <String, Nodes>. The idea of having Nodes as a value of key String (vertex label) is to sufficiently irritate through the HashMap to get the Node object of a vertex label. In the Node class, contains a LinkLinked list from java.io to store all the pairs of a node in the graph. The average time complexity of a HashMap is balanced because it consistently produces o(1) with a combination of a single linked list, whereby the average time complexity of it is o(1) for insertion and deletion. However, the linked list has its cons when indexing and searching with an average time of o(n). To improve the implemented graph of an incidence matrix, a set of a 2D array to construct the graph and LinkedList of MyPairs object to efficiently search and store a pair object of a vertex.