# Major assignment 2: Your submission

This is your assignment template for [CompX Major assignment 2](https://courses.edx.org/courses/course-v1:AdelaideX+CompX+3T2018/courseware/79355773219d46ce8241e39d7cdce010/a97f27d67b97489aacee42f78ee03357/1?activate_block_id=block-v1%3AAdelaideX%2BCompX%2B3T2018%2Btype%40vertical%2Bblock%40a84c6f7dfc2148ca9535b6520818daa1). Save this document on our local machine and include all of your work within the relevant sections. Once you’ve completed all five parts of the assignment, upload the document via the submission area on the “[Submit your assignment](https://courses.edx.org/courses/course-v1:AdelaideX+CompX+3T2018/courseware/79355773219d46ce8241e39d7cdce010/a97f27d67b97489aacee42f78ee03357/14?activate_block_id=block-v1%3AAdelaideX%2BCompX%2B3T2018%2Btype%40vertical%2Bblock%40022fb40d796042f294a4a9c9cff714ed)” page at the end of Major assignment 2.

# You will need to:

# Include both the answer to the question, and The code as required.

# Your answer will assist the University of Adelaide academic staff member assess your code submission. The point(s) value at the end of each question is only for your code as you will have received points for correctly answering the question in the course (on edX).

**Quick links:**

[Major assignment 2: Part 1](#_Major_assignment_2:_1)

[Major assignment 2: Part 2](#_Major_assignment_1:_1)

[Major assignment 2: Part 3](#_Major_assignment_1:_2)

[Major assignment 2: Part 4](#_Major_assignment_1:_3)

[Major assignment 2: Part 5](#_Major_assignment_2:)

# Major assignment 2: Part 1

1. Read in the flights dataset from the NYC flights package [1 point]

Your relevant code goes here:

    /\*\*

     \* Populate the with data read from a CSV file.

     \* @param path The path to the CSV to read the data from.

     \*/

    public void loadData(String path) {

        // initialize the columns and rows

        this.columns = new ArrayList<String>();

        this.rows = new ArrayList<TableRow>();

        // open the file for reading

        File f = new File(path);

        try {

            Scanner sc = new Scanner(f);

            if (sc.hasNextLine()) {

                // process the header row

                String line = sc.nextLine();

                // yes this is a total cheat, but not worth the effort when this is a bad way to read csv files anyway

                String columns[] = line.replace("\"", "").split(",");

                for (String column : columns) {

                    this.addColumn(column);

                }

                // load the table rows

                while( sc.hasNextLine() ) {

                    line = sc.nextLine();

                    String values[] = line.replace("\"", "").split(",");

                    // check for NA values when the row should be exluded

                    boolean include = true;

                    for (String value : values) {

                        if (value.equals("NA")) {

                            include = false;

                            break;

                        }

                    }

                    // add the row to the dataset

                    if (include) {

                        TableRow row = this.addRow();

                        for (int i = 0; i < values.length; i++) {

                            row.setString(i, values[i]);

                        }

                    }

                }

            }

        } catch (FileNotFoundException ex) {

            System.out.println("File " + f + " not found.");

        }

    }

1. Provide evidence of how you were able to count the number of flights leaving EWR, JFK and LGA [1 point]

Your answers to number of flights leaving EWR, JFK and LGA go here:

|  |
| --- |
| Flights leaving EWR: 117127  Flights leaving JFK: 109079  Flights leaving LGA: 10114 |

Your relevant code goes here:

    /\*\*

     \*  Finds the rows in the Table that contain the value provided, and returns references to those rows.

     \* @param value The value to match.

     \* @param columnName Title of the column to search.

     \* @return Returns an ArrayList of rows matching the criteria.

     \*/

    public ArrayList<TableRow> findRows(String value, String columnName) {

        ArrayList<TableRow> results = new ArrayList<TableRow>();

        for (TableRow row : this.rows) {

            if (row.getString(columnName).equals(value)) {

                results.add(row);

            }

        }

        return results;

    }

}

        // find the number of flights leaving airports

        String[] origins = {"EWR", "JFK", "LGA"};

        for (String origin : origins) {

            System.out.println(String.format(

                "Flights leaving %s: %s",

                origin,

                flights.findRows(origin, "origin").size()));

        }

1. Provide evidence of how you were able to calculate the minimum distance, median, mean and maximum distance of flights leaving LGA [1 points]

Your answers to minimum distance, median, mean and maximum distance go here:

|  |
| --- |
| LGA min : 96.0  LGA median : 762.0  LGA mean : 784.7561103421001  LGA max : 1620.0 |

Your relevant code goes here:

    /\*\*

     \* Determine the max value from a set of data points.

     \*

     \* @param data The float array of values to determine the maximum value for.

     \* @return Return the max value.

     \*/

    public static double getMax(double data[]) {

        double max = data[0];

        for (int i = 0; i < data.length; i++) {

            if (data[i] > max) {

                max = data[i];

            }

        }

        return max;

    }

    /\*

     \* Determine the max value from a set of data points.

     \*/

    public static double getMin(double data[]) {

        double min = data[0];

        for (int i = 0; i < data.length; i++) {

            if (data[i] < min) {

                min = data[i];

            }

        }

        return min;

    }

    /\*

     \* Calculate the mean for the data points.

     \*/

    public static double getMean(double[] data) {

        double total = 0;

        // calculate the sum of the data points

        for (int i = 0; i < data.length; i++) {

            total += data[i];

        }

        return total / data.length;

    }

    /\*

     \* Determine the data point median

     \*/

    public static double getMedian(double[] data) {

        Arrays.sort(data);

        int middle = data.length / 2;

        if (data.length % 2 == 0) {

            return (data[middle - 1] + data[middle]) / 2.0;

        } else {

            return data[middle];

        }

    }

        System.out.println("\n--- Part 1: Question 4-7");

        // get the LGA rows

        Table lgaTable = new Table(flights.findRows("LGA", "origin"));

        // get distance as a double array for processing

        double[] lgaDistance = lgaTable.toDoubleArray("distance");

        System.out.println("LGA min    : " + Flights.getMin(lgaDistance));

        System.out.println("LGA median : " + Flights.getMedian(lgaDistance));

        System.out.println("LGA mean   : " + Flights.getMean(lgaDistance));

        System.out.println("LGA max    : " + Flights.getMax(lgaDistance));

# Major assignment 2: Part 2

1. Analyse the different aircraft, paying attention to the tailnum field, in order to work out how many different aircraft operated for three airlines: UA, HA and B6 [1 point]

Your answers to how many aircraft operated for UA, HA and B6 go here:

|  |
| --- |
| Unique Tailnums for UA: 620  Unique Tailnums for HA: 14  Unique Tailnums for B6: 193 |

Upload the entire code for this part, with specific reference to the output for each airline:

    /\*\*

     \* Get the unqiue values a column contains.

     \* @param columnName The name to retrieve the list of unique values for.

     \* @return A String array containing unique values.

     \*/

    String[] getUnique(String columnName) {

        // get the array of values for the column

        String[] values = this.toArray(columnName);

        List<String> valuesList = Arrays.asList(values);

        // use a hash set to get the unique values

        HashSet<String> valueSet = new HashSet<String>();

        valueSet.addAll(valuesList);

        return valueSet.toArray(new String[0]);

    }

}

        System.out.println("\n--- Part 2: Question 1-3");

        String[] carriers = {"UA", "HA", "B6"};

        for (String carrier : carriers) {

            // get the flights operated by the carrier

            Table carrierTable = new Table(flights.findRows(carrier, "carrier"));

            System.out.println(String.format(

                "Unique Tailnums for %s: %s",

                carrier,

                carrierTable.getUnique("tailnum").length));

        }

2. Analyse B6, paying attention to how many different destinations B6 flew to in December [1 point]

Your answer to how many different destinations B6 flew to goes here:

|  |
| --- |
| B6 unique Destinations (December): 39 |

Your relevant code goes here:

        System.out.println("\n--- Part 2: Question 3");

        Table b6Table = new Table(flights.findRows("B6", "carrier"));

        Table b6December = new Table(b6Table.findRows("12", "month"));

        System.out.println("B6 unique Destinations (December): " + b6December.getUnique("dest").length);

3. Analyse LGA, paying attention to the airports listed in the dataset that cannot be reached directly from LGA by taking a single flight [1 point]

Your answer to how many of the airports cannot be reached directly from LGA by taking a single flight goes here:

|  |
| --- |
| Total destinations with no direct flight from LGA: 36 |

Your code goes here:

        System.out.println("\n--- Part 2: Question 5");

        // get the total number of destinations

        int destTotal = flights.getUnique("dest").length;

        // get the destinations of LGA

        int destLGA = new Table(

            flights.findRows("LGA", "origin"))

            .getUnique("dest")

            .length;

        // show the unreachable destinations count

        System.out.println("Total destinations with no direct flight from LGA: " + (destTotal - destLGA));

# Major assignment 2: Part 3

1. Identify how many existing flights will get re-allocated to ECI. Note: Round the number of flights [2 points]

Your answer to how many existing flights will get re-allocated to ECI goes here:

*This requirement is very ambiguous, I would actually go back to the product owner and the team for clarification since there are multiple ways to interpret this. I went for a simple mathematical approach to at least show one approach to estimate the number: 4/13 flights will be re-allocated to ECI.*

|  |
| --- |
| Re-allocations: 100722 |

Your code goes here:

    /\*\*

     \* Estimate how many flights will be re-scheduled when ECI is added.

     \* This is based on the simple realization that given the number of runways about 30% of flights will be send to ECI.

     \* @return The estimated number of flight changes.

     \*/

    public long estimateFlightChanges() {

        // calculate the ratio of flights ECI should receive

        int totalRunways = 3 + 2 + 4 + 4;

        double eciRatio = 4.0 / totalRunways;

        // get the total number of flights

        int totalFlights = flights.getRowCount();

        // calculate the number of flight changes

        double flightChanges = eciRatio \* totalFlights;

        // return the result

        return Math.round(flightChanges);

    }

        FlightScheduler scheduler = new FlightScheduler();

        scheduler.loadData("section10/Flights.csv");

        // calculate flight re-allocation

        System.out.println("Re-allocations: " + scheduler.estimateFlightChanges());

2. Using HashMaps or HashSets, write a constructor, FlightScheduler(), for the class FlightScheduler [2 points]

Your code goes here:

    protected Flights flights;

    protected Map<String, TableRow> clusteredFlightIndex;

    /\*\*

     \* Initialize a default instance of the class.

     \*/

    public FlightScheduler() {

        this.flights = new Flights();

        this.clusteredFlightIndex = new HashMap<String, TableRow>();

    }

3. Write a function to load the data for the class FlightScheduler with the following signature:  
void loadData(String flightDataFile) [2 points]

Your code goes here:

    /\*\*

     \* Read flight data from a CSV file.

     \* @param flightDataFile The path to the CSV file to load.

     \*/

    public void loadData(String flightDataFile) {

        // initilize this variable again in case of a second load

        this.clusteredFlightIndex = new HashMap<String, TableRow>();

        // load the data from CSV

        Table sourceData = new Flights(flightDataFile);

        // create the data table with the required additional column

        this.flights = new Flights();

        for (String columnName : sourceData.columns) {

            this.flights.addColumn(columnName);

        }

        // add the new column

        this.flights.addColumn("orignal\_origin");

        // add the source rows and initialize the orignal\_origin column

        for (TableRow row : sourceData.rows) {

            TableRow newRow = this.flights.addRow();

            // add the original data

            for (String columnName : row.columns) {

                newRow.setString(columnName, row.getString(columnName));

            }

            // initialize the orignal\_origin column

            newRow.setString("orignal\_origin", null);

            // add the row to the index

            this.addToIndex(newRow);

        }

    }

    /\*\*

     \* Get the clustered key for a fligh in the index.

     \* @param day The day of the flight.

     \* @param month The flight month.

     \* @param year The year of the flight.

     \* @param flightCode The flight code.

     \* @return Return the clustered key as a string.

     \*/

    protected String getClusterKey(String day, String month, String year, String flightCode) {

        return String.format("%s\_%s\_%s\_%s",

            year,

            month,

            day,

            flightCode

        );

    }

    /\*\*

     \* Add a row to the clustered index.

     \* @param row The row to add to the index.

     \*/

    protected void addToIndex(TableRow row) {

        String clusteredKey = this.getClusterKey(

            row.getString("day"),

            row.getString("month"),

            row.getString("year"),

            row.getString("flight"));

        // add the row to the index

        this.clusteredFlightIndex.put(clusteredKey, row);

    }

4. Write a function to reallocate flights for the class FlightScheduler with the following signature:  
void reallocate (int day, int month, int year, String flightCode) [2 points]

Your code goes here:

    /\*\*

     \* Re-allocate a flight to ECI.

     \* @param day The day of the flight.

     \* @param month The flight month.

     \* @param year The year of the flight.

     \* @param flightCode The flight code.

     \*/

    public void reallocate(int day, int month, int year, String flightCode) {

        // get the clustered key

        String clusteredKey = this.getClusterKey(

            String.valueOf(day),

            String.valueOf(month),

            String.valueOf(year),

            flightCode);

        // get the row to update

        TableRow row = this.clusteredFlightIndex.get(clusteredKey);

        // set the new origin

        row.setString("orignal\_origin", row.getString("origin"));

        row.setString("origin", "ECI");

    }

5. Write a function to check if a flight is moved for the class FlightScheduler with the following signature:  
boolean check(int day, int month, int year, String flightCode)  
[2 points]

Your code goes here:

    /\*\*

     \* Check if a flight is moved to ECI

     \* @param day The day of the flight.

     \* @param month The flight month.

     \* @param year The year of the flight.

     \* @param flightCode The flight code.

     \* @return Returns true if the flight has been moved, false otherwise.

     \*/

    public boolean check(int day, int month, int year, String flightCode) {

        // get the clustered key

        String clusteredKey = this.getClusterKey(

            String.valueOf(day),

            String.valueOf(month),

            String.valueOf(year),

            flightCode);

        // get the row to update

        TableRow row = this.clusteredFlightIndex.get(clusteredKey);

        return row.getString("orignal\_origin") != null;

    }

# Major assignment 2: Part 4

1. Identify how many different airports you can reach from EWR by taking two (2) flights. Note: This means that you need to take one flight from EWR to airport X, and then take a flight from airport X to somewhere else   
[2 points]

Your answer to how many different airports can be reached from EWR by taking two (2) flights goes here:

|  |
| --- |
| --- Two Flights:  [EWR, LGA, JFK]  3 |

Your code goes here:

/\*\*

 \* FlightGraph --- Inherits from the Graph class to represents the fligts dataset.

 \*/

public class FlightGraph extends Graph {

    protected Flights flights;

    /\*\*

     \* Initialize a new instance of the class.

     \*/

    public FlightGraph() {

        super();

        String sourcePath = "section10/Flights.csv";

        this.flights = new Flights(sourcePath);

    }

    /\*\*

     \* Get a node from the graph, if it does not exist, create it.

     \* @param name The name of the node.

     \* @return Return the graph node.

     \*/

    public Node getNodeWithCreate(String name) {

        // create the node if required

        if (!this.nodes.containsKey(name)) {

            this.addNode(name);

        }

        // return the node

        return this.getNode(name);

    }

    /\*\*

     \* Construct a cyclical graph from the flights dataset.

     \*/

    public void constructUndirectedGraph() {

        for (int i = 0; i < this.flights.getRowCount(); i++) {

            // get the row and values

            TableRow row = this.flights.getRow(i);

            String origin = row.getString("origin");

            String dest = row.getString("dest");

            // get the nodes to make sure they exist

            this.getNodeWithCreate(origin);

            this.getNodeWithCreate(dest);

            // add the edges

            if (!this.hasEdge(origin, dest)) this.addEdge(origin, dest);

            if (!this.hasEdge(dest, origin)) this.addEdge(dest, origin);

        }

    }

    /\*\*

     \* Get the child nodes names of a node.

     \* @param name The name of the node to get the names for.

     \* @return Return a list of child node names.

     \*/

    ArrayList<String> getChildNodeNames(String name) {

        ArrayList<String> childNames = new ArrayList<String>();

        ArrayList<Node> childNodes = this.getChildNodes(name);

        for (Node node : childNodes) {

            childNames.add(node.name);

        }

        return childNames;

    }

     /\*\*

      \* Recursively get a list of unique nodes for the nodes in the parent list.

      \* @param parents The partents to return nodes for.

      \* @param level The debt to search to. When 0 simply return the unique values from the partent list.

      \* @return The list of unique nodes.

      \*/

    public HashSet<String> getUniqueChildren(HashSet<String> parents, int level) {

        if (level == 0) {

            return parents;

        }

        else {

            level--;

            HashSet<String> childSet = new HashSet<String>();

            // get the set of children for the parent nodes

            for (String parent : parents) {

                childSet.addAll(this.getChildNodeNames(parent));

            }

            return this.getUniqueChildren(childSet, level);

        }

    }

        FlightGraph graph = new FlightGraph();

        graph.constructUndirectedGraph();

        System.out.println("--- Two Flights:");

        System.out.println(graph.getUniqueChildren(ewrList, 2));

        System.out.println(graph.getUniqueChildren(ewrList, 2).size());

2. Identify how many different airports you can reach from EWR by taking three (3) flights   
[2 points]

Your answer to how many different airports can be reached from EWR by taking three (3) flights goes here:

|  |
| --- |
| --- Three Flights:  [JAX, TVC, HNL, CLT, STL, XNA, SLC, STT, DFW, SDF, MHT, CMH, OMA, EYW, MIA, LGB, MYR, PWM, IAD, SMF, SEA, HOU, IAH, CVG, RSW, TPA, BTV, FLL, SNA, BDL, ABQ, BUF, PHL, SFO, BUR, PHX, RDU, MKE, MSN, ACK, MSP, TYS, MCI, MSY, BNA, ALB, MCO, ATL, PIT, DAY, LAS, LAX, MTJ, ILM, CHO, CHS, BWI, PBI, AUS, DSM, MDW, DCA, PSE, CAE, BOS, ROC, CAK, BGR, PSP, SYR, MEM, ANC, AVL, OAK, IND, ORD, SAN, ORF, DTW, GRR, BHM, SAT, SAV, CRW, SJC, MVY, HDN, SRQ, BQN, SBN, SJU, EGE, GSO, GSP, PDX, OKC, TUL, DEN, JAC, RIC, BZN, CLE, LEX, PVD]  104 |

Your code goes here:

        System.out.println("--- Three Flights:");

        System.out.println(graph.getUniqueChildren(ewrList, 3));

        System.out.println(graph.getUniqueChildren(ewrList, 3).size());

# Major assignment 2: Part 5

1. Based on the assumptions outlined within the scenario, identify the maximum number of flights that you could have taken in 2013 [6 points]

Your answer to the maximum number of flights you can take in 2013 goes here:

I sincerely apologize for not completing the section of the assignment, the amount of time I would have to put into this section would be disproportionate.

Your code goes here:

# Appendix I: Flights.java

/\*\*

 \* Major Assignment 02 --- NYC Flights Analysis

 \* @author                 Johannes Foulds

 \*/

package section10;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.Scanner;

import java.util.Arrays;

import java.util.HashSet;

import java.util.List;

/\*\*

 \* TableRow --- A TableRow object represents a single row of data values, stored in columns, from a table.

 \*/

class TableRow {

    protected ArrayList<String> columns;

    protected String[]  values;

    /\*\*

     \* Create a new instance of a TableRow.

     \* @param columns The columns of the row.

     \*/

    public TableRow(ArrayList<String> columns) {

        this.columns = columns;

        this.values = new String[this.columns.size()];

    }

    /\*\*

     \* Set the string value of the column at the specified index.

     \* @param column The index of the column to update.

     \* @param value The value to set the column to.

     \*/

    public void setString(int column, String value) {

        this.values[column] = value;

    }

    /\*\*

     \* Set the string value of the column at the specified index.

     \* @param columnName The title of the column to update.

     \* @param value The value to set the column to.

     \*/

    public void setString(String columnName, String value) {

        int column = this.columns.indexOf(columnName);

        this.setString(column, value);

    }

    /\*\*

     \* Get the string value from a row.

     \* @param columnName The name of the column to get the value for.

     \* @return Returns the value of the column.

     \*/

    public String getString(String columnName) {

        int column = this.columns.indexOf(columnName);

        return this.values[column];

    }

    /\*\*

     \* Get a double value from the specified column

     \* @param columnName The name of the column to get the value for.

     \* @return Returns the value of the column.

     \*/

    public double getDouble(String columnName) {

        int column = this.columns.indexOf(columnName);

        return Double.parseDouble(this.values[column]);

    }

}

/\*\*

 \* Table --- Table objects store data with multiple rows and columns.

 \*/

class Table {

    protected ArrayList<String> columns;

    protected ArrayList<TableRow> rows;

    /\*\*

     \* Create a default instance of the class.

     \*/

    public Table() {

        this.columns = new ArrayList<String>();

        this.rows = new ArrayList<TableRow>();

    }

    /\*\*

     \* Create a new table from an ArrayList of TableRows.

     \* @param rows The rows to initialize the table with.

     \*/

    public Table(ArrayList<TableRow> rows) {

        this();

        if (rows.size() > 0) {

            this.columns = rows.get(0).columns;

            this.rows = rows;

        }

    }

    /\*\*

     \* Add a new table column.

     \* @param title The column title.

     \*/

    public void addColumn(String title) {

        this.columns.add(title);

    }

    /\*\*

     \* Add a new table row.

     \* @return Return the new table row.

     \*/

    public TableRow addRow() {

        TableRow row = new TableRow(this.columns);

        this.rows.add(row);

        return row;

    }

    /\*\*

     \* Get the number of table tows.

     \* @return Return the total number of rows in the table.

     \*/

    public int getRowCount() {

        return this.rows.size();

    }

    /\*\*

     \* Get a row at a specific index.

     \* @param row The row index of the row to retrieve.

     \* @return Return the row at the row index.

     \*/

    public TableRow getRow(int row) {

        return this.rows.get(row);

    }

    /\*\*

     \*  Gets all rows from the table.

     \* @return All the rows in the table.

     \*/

    public ArrayList<TableRow> rows() {

        return this.rows();

    }

    /\*\*

     \*  Finds the rows in the Table that contain the value provided, and returns references to those rows.

     \* @param value The value to match.

     \* @param columnName Title of the column to search.

     \* @return Returns an ArrayList of rows matching the criteria.

     \*/

    public ArrayList<TableRow> findRows(String value, String columnName) {

        ArrayList<TableRow> results = new ArrayList<TableRow>();

        for (TableRow row : this.rows) {

            if (row.getString(columnName).equals(value)) {

                results.add(row);

            }

        }

        return results;

    }

    /\*\*

     \* Get a column as a string array.

     \* @param columnName The column to convert to an array.

     \* @return Returns a String array with all values in the column.

     \*/

    public String[] toArray(String columnName) {

        String[] result = new String[this.getRowCount()];

        for (int i = 0; i < this.getRowCount(); i++) {

            result[i] = this.getRow(i).getString(columnName);

        }

        return result;

    }

    /\*\*

     \* Get a column as a double array.

     \* @param columnName The column to convert to an array.

     \* @return Returns a double array with all values in the column.

     \*/

    public double[] toDoubleArray(String columnName) {

        double[] result = new double[this.getRowCount()];

        for (int i = 0; i < this.getRowCount(); i++) {

            result[i] = this.getRow(i).getDouble(columnName);

        }

        return result;

    }

    /\*\*

     \* Get the unqiue values a column contains.

     \* @param columnName The name to retrieve the list of unique values for.

     \* @return A String array containing unique values.

     \*/

    String[] getUnique(String columnName) {

        // get the array of values for the column

        String[] values = this.toArray(columnName);

        List<String> valuesList = Arrays.asList(values);

        // use a hash set to get the unique values

        HashSet<String> valueSet = new HashSet<String>();

        valueSet.addAll(valuesList);

        return valueSet.toArray(new String[0]);

    }

}

/\*\*

 \* Flights --- Inherit from the Table class and implement functionality specific to flights.

 \*/

public class Flights extends Table {

    /\*\*

     \* Create a default instance of the class.

     \*/

    public Flights() {

    }

    /\*\*

     \* Create a new instance of the class and populate it with data read from a CSV file.

     \* @param path The path to the CSV to read the data from.

     \*/

    public Flights(String path) {

        super();

        this.loadData(path);

    }

    /\*\*

     \* Populate the with data read from a CSV file.

     \* @param path The path to the CSV to read the data from.

     \*/

    public void loadData(String path) {

        // initialize the columns and rows

        this.columns = new ArrayList<String>();

        this.rows = new ArrayList<TableRow>();

        // open the file for reading

        File f = new File(path);

        try {

            Scanner sc = new Scanner(f);

            if (sc.hasNextLine()) {

                // process the header row

                String line = sc.nextLine();

                // yes this is a total cheat, but not worth the effort when this is a bad way to read csv files anyway

                String columns[] = line.replace("\"", "").split(",");

                for (String column : columns) {

                    this.addColumn(column);

                }

                // load the table rows

                while( sc.hasNextLine() ) {

                    line = sc.nextLine();

                    String values[] = line.replace("\"", "").split(",");

                    // check for NA values when the row should be exluded

                    boolean include = true;

                    for (String value : values) {

                        if (value.equals("NA")) {

                            include = false;

                            break;

                        }

                    }

                    // add the row to the dataset

                    if (include) {

                        TableRow row = this.addRow();

                        for (int i = 0; i < values.length; i++) {

                            row.setString(i, values[i]);

                        }

                    }

                }

            }

        } catch (FileNotFoundException ex) {

            System.out.println("File " + f + " not found.");

        }

    }

    /\*\*

     \* Determine the max value from a set of data points.

     \*

     \* @param data The float array of values to determine the maximum value for.

     \* @return Return the max value.

     \*/

    public static double getMax(double data[]) {

        double max = data[0];

        for (int i = 0; i < data.length; i++) {

            if (data[i] > max) {

                max = data[i];

            }

        }

        return max;

    }

    /\*

     \* Determine the max value from a set of data points.

     \*/

    public static double getMin(double data[]) {

        double min = data[0];

        for (int i = 0; i < data.length; i++) {

            if (data[i] < min) {

                min = data[i];

            }

        }

        return min;

    }

    /\*

     \* Calculate the mean for the data points.

     \*/

    public static double getMean(double[] data) {

        double total = 0;

        // calculate the sum of the data points

        for (int i = 0; i < data.length; i++) {

            total += data[i];

        }

        return total / data.length;

    }

    /\*

     \* Determine the data point median

     \*/

    public static double getMedian(double[] data) {

        Arrays.sort(data);

        int middle = data.length / 2;

        if (data.length % 2 == 0) {

            return (data[middle - 1] + data[middle]) / 2.0;

        } else {

            return data[middle];

        }

    }

    /\*\*

     \* This is the entry point to the application for answering the assignment questions.

     \* @param args

     \*/

    public static void main(String[] args) {

        String sourcePath = "section10/Flights.csv";

        // create a new instance of the class

        Flights flights = new Flights(sourcePath);

        System.out.println("Rows Read: " + flights.getRowCount());

        // get a test row

        TableRow testRow = flights.getRow(0);

        System.out.println("0 - tailnum: " + testRow.getString("tailnum"));

        // find the number of flights leaving airports

        String[] origins = {"EWR", "JFK", "LGA"};

        for (String origin : origins) {

            System.out.println(String.format(

                "Flights leaving %s: %s",

                origin,

                flights.findRows(origin, "origin").size()));

        }

        System.out.println("\n--- Part 1: Question 4-7");

        // get the LGA rows

        Table lgaTable = new Table(flights.findRows("LGA", "origin"));

        // get distance as a double array for processing

        double[] lgaDistance = lgaTable.toDoubleArray("distance");

        System.out.println("LGA min    : " + Flights.getMin(lgaDistance));

        System.out.println("LGA median : " + Flights.getMedian(lgaDistance));

        System.out.println("LGA mean   : " + Flights.getMean(lgaDistance));

        System.out.println("LGA max    : " + Flights.getMax(lgaDistance));

        System.out.println("\n--- Part 2: Question 1-3");

        String[] carriers = {"UA", "HA", "B6"};

        for (String carrier : carriers) {

            // get the flights operated by the carrier

            Table carrierTable = new Table(flights.findRows(carrier, "carrier"));

            System.out.println(String.format(

                "Unique Tailnums for %s: %s",

                carrier,

                carrierTable.getUnique("tailnum").length));

        }

        System.out.println("\n--- Part 2: Question 4");

        Table b6Table = new Table(flights.findRows("B6", "carrier"));

        Table b6December = new Table(b6Table.findRows("12", "month"));

        System.out.println("B6 unique Destinations (December): " + b6December.getUnique("dest").length);

        System.out.println("\n--- Part 2: Question 5");

        // get the total number of destinations

        int destTotal = flights.getUnique("dest").length;

        // get the destinations of LGA

        int destLGA = new Table(

            flights.findRows("LGA", "origin"))

            .getUnique("dest")

            .length;

        // show the unreachable destinations count

        System.out.println("Total destinations with no direct flight from LGA: " +

            (destTotal - destLGA));

    }

}

# Appendix II: FlightScheduler.java

/\*\*

 \* Major Assignment 02 --- NYC Flights Analysis: Part 3

 \* @author                 Johannes Foulds

 \*/

package section10;

import java.util.HashMap;

import java.util.Map;

/\*\*

 \* FlightScheduler --- Implement a flight scheduled.

 \*/

public class FlightScheduler {

    protected Flights flights;

    protected Map<String, TableRow> clusteredFlightIndex;

    /\*\*

     \* Initialize a default instance of the class.

     \*/

    public FlightScheduler() {

        this.flights = new Flights();

        this.clusteredFlightIndex = new HashMap<String, TableRow>();

    }

    /\*\*

     \* Read flight data from a CSV file.

     \* @param flightDataFile The path to the CSV file to load.

     \*/

    public void loadData(String flightDataFile) {

        // initilize this variable again in case of a second load

        this.clusteredFlightIndex = new HashMap<String, TableRow>();

        // load the data from CSV

        Table sourceData = new Flights(flightDataFile);

        // create the data table with the required additional column

        this.flights = new Flights();

        for (String columnName : sourceData.columns) {

            this.flights.addColumn(columnName);

        }

        // add the new column

        this.flights.addColumn("orignal\_origin");

        // add the source rows and initialize the orignal\_origin column

        for (TableRow row : sourceData.rows) {

            TableRow newRow = this.flights.addRow();

            // add the original data

            for (String columnName : row.columns) {

                newRow.setString(columnName, row.getString(columnName));

            }

            // initialize the orignal\_origin column

            newRow.setString("orignal\_origin", null);

            // add the row to the index

            this.addToIndex(newRow);

        }

    }

    /\*\*

     \* Get the clustered key for a fligh in the index.

     \* @param day The day of the flight.

     \* @param month The flight month.

     \* @param year The year of the flight.

     \* @param flightCode The flight code.

     \* @return Return the clustered key as a string.

     \*/

    protected String getClusterKey(String day, String month, String year, String flightCode) {

        return String.format("%s\_%s\_%s\_%s",

            year,

            month,

            day,

            flightCode

        );

    }

    /\*\*

     \* Add a row to the clustered index.

     \* @param row The row to add to the index.

     \*/

    protected void addToIndex(TableRow row) {

        String clusteredKey = this.getClusterKey(

            row.getString("day"),

            row.getString("month"),

            row.getString("year"),

            row.getString("flight"));

        // add the row to the index

        this.clusteredFlightIndex.put(clusteredKey, row);

    }

    /\*\*

     \* Re-allocate a flight to ECI.

     \* @param day The day of the flight.

     \* @param month The flight month.

     \* @param year The year of the flight.

     \* @param flightCode The flight code.

     \*/

    public void reallocate(int day, int month, int year, String flightCode) {

        // get the clustered key

        String clusteredKey = this.getClusterKey(

            String.valueOf(day),

            String.valueOf(month),

            String.valueOf(year),

            flightCode);

        // get the row to update

        TableRow row = this.clusteredFlightIndex.get(clusteredKey);

        // set the new origin

        row.setString("orignal\_origin", row.getString("origin"));

        row.setString("origin", "ECI");

    }

    /\*\*

     \* Check if a flight is moved to ECI

     \* @param day The day of the flight.

     \* @param month The flight month.

     \* @param year The year of the flight.

     \* @param flightCode The flight code.

     \* @return Returns true if the flight has been moved, false otherwise.

     \*/

    public boolean check(int day, int month, int year, String flightCode) {

        // get the clustered key

        String clusteredKey = this.getClusterKey(

            String.valueOf(day),

            String.valueOf(month),

            String.valueOf(year),

            flightCode);

        // get the row to update

        TableRow row = this.clusteredFlightIndex.get(clusteredKey);

        return row.getString("orignal\_origin") != null;

    }

    /\*\*

     \* Estimate how many flights will be re-scheduled when ECI is added.

     \* This is based on the simple realization that given the number of runways about 30% of flights will be send to ECI.

     \* @return The estimated number of flight changes.

     \*/

    public long estimateFlightChanges() {

        // calculate the ratio of flights ECI should receive

        int totalRunways = 3 + 2 + 4 + 4;

        double eciRatio = 4.0 / totalRunways;

        // get the total number of flights

        int totalFlights = flights.getRowCount();

        // calculate the number of flight changes

        double flightChanges = eciRatio \* totalFlights;

        // return the result

        return Math.round(flightChanges);

    }

    /\*\*

     \* The main function for testing and debugging.

     \* @param args Command line arguments.

     \*/

    public static void main(String[] args) {

        FlightScheduler scheduler = new FlightScheduler();

        scheduler.loadData("section10/Flights.csv");

        // calculate flight re-allocation

        //System.out.println("Re-allocations: " + scheduler.estimateFlightChanges());

        // reallocate a flight

        System.out.println(scheduler.check(25, 2, 2013, "743"));

        scheduler.reallocate(10, 7, 2013, "3910");

        System.out.println(scheduler.check(10, 7, 2013, "3910"));

        System.out.println(scheduler.check(25, 2, 2013, "743"));

    }

}

# Appendix II: FlightGraph.java

package section10;

import java.util.ArrayList;

import java.util.HashSet;

import java.util.Map;

import java.util.Set;

import java.util.TreeMap;

import java.util.UUID;

/\*\*

 \* Graph --- This class inplement a graph structure with child nodes.

 \*/

class Graph {

    protected Map<String, Node> nodes;

    protected Edges edges;

    /\*\*

     \* Create a default instance of the class.

     \*/

    public Graph() {

        this.nodes = new TreeMap<String, Node>();

        this.edges = new Edges();

    }

    /\*\*

     \* Add a node to the graph network.

     \* @param name The name of the node.

     \*/

    public void addNode(String name) {

        // create the new node and add it to the nodes collection

        Node node = new Node(name);

        this.nodes.put(name, node);

    }

    /\*\*

     \* Remove the node with a given name from the graph network.

     \* @param name The name of the node to remove.

     \*/

    public void removeNode(String name) {

        // validate that the node exist

        if (!this.nodes.containsKey(name)) throw new IndexOutOfBoundsException("The node '" + name + "' does not exist.");

        Node node = this.nodes.get(name);

        // remove the node from the edges

        this.edges.removeNode(node);

        // remove the node

        this.nodes.remove(name);

    }

    /\*\*

     \* Returns the Node object with a given name from the graph network if it exists. Returns null otherwise.

     \* @param name The name of the node tp get.

     \* @return The node if it exists, null otherwise.

     \*/

    public Node getNode(String name) {

        // validate that the node exist

        if (!this.nodes.containsKey(name)) return null;

        return this.nodes.get(name);

    }

    /\*\*

     \* Connect two nodes with given names together if they exist.

     \* @param node1 The first node to create the edge for.

     \* @param node2 The node to connect the fist to.

     \*/

    public void addEdge(String node1, String node2) {

        // validate that the nodes exist

        if (!this.nodes.containsKey(node1)) throw new IndexOutOfBoundsException("The node '" + node1 + "' does not exist.");

        if (!this.nodes.containsKey(node2)) throw new IndexOutOfBoundsException("The node '" + node2 + "' does not exist.");

        // add the edge

        this.edges.addEdge(this.nodes.get(node1), this.nodes.get(node2));

    }

    /\*\*

     \* Determine if an edge exists between two nodes.

     \* @param node1 The first node to check for an edge.

     \* @param node2 The second node to check for the existence of an edge.

     \* @return Returns a boolean value; true if there is an edge from node1 to node2, false otherwise

     \*/

    public boolean hasEdge(String node1, String node2) {

        // validate that the nodes exist

        if (!this.nodes.containsKey(node1)) return false;

        if (!this.nodes.containsKey(node2)) return false;

        // determine if an edge exist for the nodes

        return this.edges.hasEdge(this.nodes.get(node1), this.nodes.get(node2));

    }

    /\*\*

     \* Remove the edge between the given nodes if one exists.

     \* @param node1 The first node to remove the edge from.

     \* @param node2 The second node of the edge.

     \*/

    public void removeEdge(String node1, String node2) {

        if (this.hasEdge(node1, node2)) {

            this.edges.removeEdge(this.nodes.get(node1), this.nodes.get(node2));

        }

    }

    /\*\*

     \* Get a klist of connected nodes for the specifided note.

     \* @param node The node to get the list for.

     \* @return Returns a list of child nodes.

     \*/

    public ArrayList<Node> getChildNodes(String node) {

        return this.edges.edges.get(node);

    }

    /\*\*

     \* Display a node and it's edged.

     \* @param node The node to display the edges for.

     \*/

    public void printStructure(String node) {

        ArrayList<Node> childNodes = this.getChildNodes(node);

        System.out.print(node);

        // print the connections

        for (Node chidlNode : childNodes) {

            System.out.print("-" + chidlNode.name);

        }

        // print the newline

        System.out.println();

    }

    /\*\*

     \* Display all of the nodes and all of the edges between nodes.

     \*/

    public void printStructure() {

        for ( Map.Entry<String, Node> nodeEntry : this.nodes.entrySet() ) {

            Node node = nodeEntry.getValue();

            System.out.println(node.toString());

        }

        // print the edges

        this.edges.printStructure();

    }

}

/\*\*

 \* Node --- This class is used to store node information used in the Graph class.

 \*/

class Node {

    public String \_id;

    public String name;

    /\*\*

     \* Create a default instance of the class.

     \*/

    public Node() {

        // assign a unique id to the node

        this.\_id = UUID.randomUUID().toString();

        this.name = null;

    }

    /\*\*

     \* Create a new instance of a now with a specific name.

     \* @param name

     \*/

    public Node(String name) {

        this();

        this.name = name;

    }

    /\*\*

     \* Display the node structure.

     \*/

    public String toString() {

        return this.name;

    }

}

/\*\*

 \* Edges -- This class represents edges in a graph network.

 \*/

class Edges {

    protected Map<String, ArrayList<Node>> edges;

    /\*\*

     \* Create a default instance of the Edges class.

     \*/

    public Edges() {

        this.edges = new TreeMap<String, ArrayList<Node>>();

    }

    /\*\*

     \* Add the edge between node11 and node2.

     \* @param node1 This node to add the edge for.

     \* @param node2 The node to connect to.

     \*/

    public void addEdge(Node node1, Node node2) {

        // add an entry for node1 if it does not already exist.

        if (!this.edges.containsKey(node1.name)) {

            this.edges.put(node1.name, new ArrayList<Node>());

        }

        // add node2 to the connections of node1

        this.edges.get(node1.name).add(node2);

    }

    /\*\*

     \* Remove a node and all its edged.

     \* @param name The the node to remove.

     \*/

    public void removeNode(Node node) {

        // remove the node entry

        this.edges.remove(node.name);

        // remove connections in the other nodes

        for ( Map.Entry<String, ArrayList<Node>> edgeEntry : this.edges.entrySet() ) {

            ArrayList<Node> connections = edgeEntry.getValue();

            if (connections.contains(node)) {

                connections.remove(node);

            }

        }

    }

    /\*\*

     \* Determine if an edge exists between two nodes.

     \* @param node1 The first node to check for an edge.

     \* @param node2 The second node to check for the existence of an edge.

     \* @return Returns a boolean value; true if there is an edge from node1 to node2, false otherwise

     \*/

    public boolean hasEdge(Node node1, Node node2) {

        // if no edges exist for node 1, then there is no edges

        if (!this.edges.containsKey(node1.name)) {

            return false;

        } else {

            ArrayList<Node> connections = this.edges.get(node1.name);

            return connections.contains(node2);

        }

    }

    /\*\*

     \* Remove the edge between the given nodes if one exists.

     \* @param node1 The first node to remove the edge from.

     \* @param node2 The second node of the edge.

     \*/

    public void removeEdge(Node node1, Node node2) {

        if (this.edges.containsKey(node1.name)) {

            ArrayList<Node> connections = this.edges.get(node1.name);

            if (connections.contains(node2)) {

                connections.remove(node2);

            }

        }

    }

    /\*\*

     \* Display the edge connections between nodes.

     \*/

    public void printStructure() {

        for ( Map.Entry<String, ArrayList<Node>> edgeEntry : this.edges.entrySet() ) {

            String nodeName = edgeEntry.getKey();

            ArrayList<Node> connections = edgeEntry.getValue();

            System.out.print(nodeName.toString());

            // print the connections

            for (Node node : connections) {

                System.out.print("-" + node.name);

            }

            // print the newline

            System.out.println();

        }

    }

}

/\*\*

 \* FlightGraph --- Inherits from the Graph class to represents the fligts dataset.

 \*/

public class FlightGraph extends Graph {

    protected Flights flights;

    /\*\*

     \* Initialize a new instance of the class.

     \*/

    public FlightGraph() {

        super();

        String sourcePath = "section10/Flights.csv";

        this.flights = new Flights(sourcePath);

    }

    /\*\*

     \* Get a node from the graph, if it does not exist, create it.

     \* @param name The name of the node.

     \* @return Return the graph node.

     \*/

    public Node getNodeWithCreate(String name) {

        // create the node if required

        if (!this.nodes.containsKey(name)) {

            this.addNode(name);

        }

        // return the node

        return this.getNode(name);

    }

    /\*\*

     \* Construct a cyclical graph from the flights dataset.

     \*/

    public void constructUndirectedGraph() {

        for (int i = 0; i < this.flights.getRowCount(); i++) {

            // get the row and values

            TableRow row = this.flights.getRow(i);

            String origin = row.getString("origin");

            String dest = row.getString("dest");

            // get the nodes to make sure they exist

            this.getNodeWithCreate(origin);

            this.getNodeWithCreate(dest);

            // add the edges

            if (!this.hasEdge(origin, dest)) this.addEdge(origin, dest);

            if (!this.hasEdge(dest, origin)) this.addEdge(dest, origin);

        }

    }

    /\*\*

     \* Get the child nodes names of a node.

     \* @param name The name of the node to get the names for.

     \* @return Return a list of child node names.

     \*/

    ArrayList<String> getChildNodeNames(String name) {

        ArrayList<String> childNames = new ArrayList<String>();

        ArrayList<Node> childNodes = this.getChildNodes(name);

        for (Node node : childNodes) {

            childNames.add(node.name);

        }

        return childNames;

    }

     /\*\*

      \* Recursively get a list of unique nodes for the nodes in the parent list.

      \* @param parents The partents to return nodes for.

      \* @param level The debt to search to. When 0 simply return the unique values from the partent list.

      \* @return The list of unique nodes.

      \*/

    public HashSet<String> getUniqueChildren(HashSet<String> parents, int level) {

        if (level == 0) {

            return parents;

        }

        else {

            level--;

            HashSet<String> childSet = new HashSet<String>();

            // get the set of children for the parent nodes

            for (String parent : parents) {

                childSet.addAll(this.getChildNodeNames(parent));

            }

            return this.getUniqueChildren(childSet, level);

        }

    }

    /\*\*

     \* The main function to test the functionality.

     \* @param args The command line arguments.

     \*/

    public static void main(String[] args) {

        FlightGraph graph = new FlightGraph();

        graph.constructUndirectedGraph();

        graph.printStructure("EWR");

        System.out.println(graph.getChildNodes("EWR").size());

        System.out.println();

        HashSet<String> ewrList = new HashSet<String>();

        ewrList.add("EWR");

        System.out.println(graph.getUniqueChildren(ewrList, 1));

        System.out.println(graph.getUniqueChildren(ewrList, 1).size());

        System.out.println("--- Two Flights:");

        System.out.println(graph.getUniqueChildren(ewrList, 2));

        System.out.println(graph.getUniqueChildren(ewrList, 2).size());

        System.out.println("--- Three Flights:");

        System.out.println(graph.getUniqueChildren(ewrList, 3));

        System.out.println(graph.getUniqueChildren(ewrList, 3).size());

    }

}