CS3310 – Fall 2015 **Asgn 6 (Shortest Path)**
Kaminski **Western Europe DrivingApp**

******************************* **OVERVIEW** ********************************

The **DrivingApp** program finds the shortest route from a start city to a destination city using Dijkstra's Minimum Cost Path (MCP) algorithm and map data for the major cities of Western Europe. Batch processing is used to determine routes for a series of start/destinations city pairs stored in the transaction file, logging results to an output file.

Two utility programs are also needed. **Setup** converts the raw map data into a more efficient-to-use format for DrivingApp to use. And **PrettyPrint** aids the developer in checking the newly created map data files.

************************** **3 SEPARATE PROGRAMS** **************************
(each in its own physically separate file)

1. **Setup** (a utility, so procedural or OOP program) BUILDS `MapGraph.bin` & `CityNameList.csv` files from the raw data file, `EuropeMapData.csv`.

2. **PrettyPrint** (a utility, so procedural or OOP program) DISPLAYS `MapGraph.bin` & `CityNameList.csv` files to `MapDataPrintout.txt` file.

3. **DrivingApp** (an OOP program) USES `MapGraph.bin` & `CityNameList.csv` and Dijkstra's (MCP) algorithm to find the shortest path for each pair of start & destination cities. Input data is in `CityPairs.csv` file, and results going to `Log.txt` file.

   DrivingApp uses 3 separate OOP classes (each in its own physically separate file):
   a. Map class – handles everything to do with `MapGraph.bin` & `CityNameList.csv` files
   b. Route class – handles everything to do with the route-finding (using Dijkstra's MCP Algorithm) and reporting its results: traceOfTargets and answer (both shortestPath & totalDistance)
   c. Log class – handles everything to do with `Log.txt` file

****************************** **6 DATA FILES** *******************************

1.	**EuropeMapData.csv**	*[MY file]*	Input to Setup
2.	**CityPairs.csv**	*[MY file]*	Input to DrivingApp
3.	**MapDataPrintout.txt**		Output from PrettyPrint
4.	**Log.txt**		Output from DrivingApp *(handled by Log class)* *[log object passed to Route class for writing to it]*
5.	**MapGraph.bin**		Output from Setup Input to PrettyPrint Input to DrivingApp *(handled by Map class)* *[map object passed to Route class's methods for roadDistance lookup, as needed]*
6.	**CityNamesList.csv**		Output from Setup Input to PrettyPrint Input to DrivingApp *(handled by Map class)* *[map object passed to Route class's methods for cityName & cityCode lookup, as needed]*

Map Data is stored in a more efficient-to-use storage structure as compared to the raw EuropeMapData file. There are 2 files:

1. `MapGraph.bin` contains:
   - N is the number of nodes (cities) in the graph, numbered 0 → N-1.
   - The map's roads (edge weights) are implemented as an ADJACENCY MATRIX. Since it's an UNDIRECTED graph, a TRIANGULAR MATRIX saves space – the lower-left triangle is used. EXTERNAL storage is used since in the future there'll be lots more cities and roads. It's a BINARY file since it's just numbers for roadDistances and N being stored.
2. `CityNameList.csv` is an unordered list of node names (10-char names, right-padded where needed) and their 3-char codes, to provider for friendlier user interaction for the DrivingApp, which uses city NAMES rather than city NUMBERS.

************************** **DrivingApp PROGRAM** **************************

```
Declare 3 objects:  map, route, log
Open CityPairs file
Loop til no more city pairs
    {   read in a city pair:  1 start city NAME & 1 destination city NAME
        ask Map's method to getCityNumber for the 2 cities' names (separately)
        ask Route's method to findMinCostPath (given the above 2 city NUMBERS)
                        [it'll need to access map and log objects]
                which will find the answer (path & distance) & the traceOfTargets
                        and provide these to log object for writing to the Log file
    }   // may have to change this to a "process-read loop structure, with priming-read" instead
Close CityPairs file
FinishUp with the 3 objects
```

********************** **3 CLASSES (used by DrivingApp)** ***********************

Log class

- public methods:
    - constructor – opens file
    - finishUp – closes file
    - displayThis(string s) – writes the string sent into it WITHOUT a <CR><LF>
    - displayThisLine(string s) – writes the string sent into it WITH a <CR><LF>

[NOTE: Setup does not write to this file – it writes to MapDataPrintout.txt file so that it can easily be printed for the developer to check it].

+ +

Map class

- all data is used DIRECTLY from the 2 FILES – it is NEVER all read into memory (EXCEPT N, which is read into memory just after opening the file)
- public methods:
    - constructor – opens files and reads in N
    - finishUp – closes files
    - getCityName(short cityNumber) – returns a string
        - direct address of `CityNameList.csv`
        - reads in the 10-char name
        - trims the right-end spaces to return a string
        - NOTE: cityNumber will ALWAYS be a valid number from 0 → N-1
    - getCityCode(short cityNumber) – returns a string
        - direct address of `CityNameList.csv`
        - reads in the 3-char name
        - NOTE: cityNumber will ALWAYS be a valid number from 0 → N-1
    - getCityNumber(string cityName) – returns a short
        - linear search of `CityNameList.csv`
        - capitalize targetCityName & cityNameInList) when comparing
        - NOTE: cityName MAY NOT find a match (e.g., Kalamazoo)
    - getRoadDistance(short cityNumber1, short cityNumber2) – returns a short
        - if 2 cityNumbers are =, their roadDistance won't be in the data file – but it's a VIRTUAL DISTANCE of 0 - so just return a 0 – OTHERWISE . . .
        - uses random access to get the designated distance – which could be
            - an actual roadDistance
            - OR "infinity" (= maxShort) meaning "no connecting road"
        - byteOffset calculation allows for headerRec (i.e., N) and calculates where in LOWER-LEFT TRIANGULAR MATRIX the appropriate distance is, based on ROW and COLUMN
        - NOTE: row > column for the lower-left triangle, so row & column could be EITHER: city1 & city2 OR city2 & city1

[NOTE: The code outside this class does NOT KNOW HOW the graph is implemented/accessed here inside this class - for example, whether things are internal or external, or whether the graph is an adjacency matrix or adjacency lists, or whether the cityNamesList is ordered or unordered or a BST or a hashTable or. . . etc.]

+ +

Route class

- contains the 3 "working-storage" (scratch) arrays
- uses Dijkstra's Minimum Cost Path algorithm *[NOTE: **<u>YOU MUST USE</u>:** **<u>Kaminski's pseudocode version</u>** of Dijkstra's Shortest Path Algorithm where you code for TRACE display]*
- public methods:
    - constructor: ???
    - finishUp: ???
    - findMinCostPath(short startNum, short destinationNum, short n, Map map, Log log)
        - this is the controller which pretty much just calls private methods:
            1. initialize3ScratchArrays(. . .)
            2. searchForPath(. . .)
            3. reportAnswer(. . .)
- 2 parts to the reported answer:
    1. total distance of the selected path
    2. list of cities in the selected path from START to DESTINATION (not vice versa)
        - use city NAMES (NOT city NUMBERS)
- the search prints out each TARGET CITY that it picks **(at the time it selects it!)**
    - NOTE: this is a TRACE of the TARGET cities (as they are selected). It is NOT just the list of cities that were INCLUDED as targets (i.e., in cityNumber order)
    - use city NAMES (NOT city NUMBERS)

[NOTE: The code outside this class does NOT KNOW what procedure was used to find the shortest path – whether it was an algorithm which calculates the answer, which algorithm was used, a search of an existing database for a stored answer, a crowd-sourced suggestion from the internet or your social network, use of your smart phone to use an app like Google maps, or. . .]

**************************** **FILE DESCRIPTIONS** ****************************

<u>MY 2 FILES</u> *[lines starting with % are comment-lines]* *[see actual files for format]*

- **<u>`EuropeMapData.csv`</u>**
- **<u>`CityPairs.csv`</u>**

+ +

<u>YOUR 4 FILES</u>

`MapGraph.bin` – graph data (N & lower-triangular adjacency matrix)

- BINARY file of all SHORTS, no field-separators, no <CR><LF>
- Header record contains just N
- Matrix data is “row-major / column-minor”
  (i.e., row changes slower, column changes faster)
- byteOffset calculation based on row & column – to be discussed in class
- 2 possible values for roadDistances:
  1. Actual value for road distance
  2. “Infinity” (i.e., maxShort) for “no road between those 2 cities”

+ +

`CityNamesList.csv` – the graph’s node (city) names

- 1 city per line: 10-char name (right-padded with spaces, if needed), a comma, the 3-char code, then <CR><LF>

+ +

`MapDataPrintout.txt` *FROM PrettyPrint*

```
Map Data:  Europe - 21 cities

00|
01| 133
02| 987 654
etc.
etc.
18| 123 --- --- 234 --- --- 345 456 --- 678 789  etc. etc.
19| 222 333 --- --- --- --- --- 888 777 --- ---  etc. etc.  555
20| --- --- 212 323 434 --- --- --- 545 656 ---  etc. etc.  989 878
   --------------------------------------------------------------------
    AMS BEL BER BRN BRU BUD COP GEN HAM LIS MAD  etc. etc.  WAR DUB LON
     00  01  02  03  04  05  06  07  08  09  10  etc. etc.   18  19  20


00 - AMS Amsterdam
01 - BEL Belgium
etc.
20 - LON London
```

NOTES

1. *Data above is NOT correct – I’m just showing the formatting.*
2. *The “etc.’s” would be filled in with data, of course.*
3. *1 distance (1103 for [10,3], in effect) is a 4-digit number, the rest are all 3-digit numbers.*
   So there’ll just be “no space” between continuous values for that entry.
4. *The diagonal data is NOT in the file itself – just print those in prettyPrint.*
5. *For “infinity’s” in the file, print “`---`“ instead, for better readability.*
6. *FOR DEMO: Use landscape, Courier Font , size 8 font, smaller margins, 2-sided printing to print this file (in WordPad) to save paper.*

+ +

`Log.txt` *FROM DrivingApp*

```
#   #   #   #   #   #   #   #   #   #   #   #
Amsterdam (0) TO Kalamazoo (-1)
ERROR - one city is NOT on this map
#   #   #   #   #   #   #   #   #   #   #   #
Kalamazoo (-1) TO Amsterdam (0)
ERROR - one city is NOT on this map
#   #   #   #   #   #   #   #   #   #   #   #
Paris (13) TO Copenhagen (6)
DISTANCE:  907
PATH:  Paris > Brussels > Amsterdam > Hamburg > Copenhagen

TRACE OF TARGETS:  BRU AMS BRN GEN HAM MAD MUN ROM BER TRI COP
# TARGETS:  11
#   #   #   #   #   #   #   #   #   #   #   #
Amsterdam (0) TO Bern (3)
DISTANCE:  558
PATH:  Amsterdam > Bern
TRACE OF TARGETS:  BRU HAM PAR MUN BRN
# TARGETS:  5
#   #   #   #   #   #   #   #   #   #   #   #
Warsaw (18) TO Warsaw (18)
DISTANCE:  0
PATH:  Warsaw
TRACE OF TARGETS: WAR
# TARGETS:  0
#   #   #   #   #   #   #   #   #   #   #   #
London (20) TO Paris (13)
DISTANCE:  ?
PATH:  SORRY - can NOT get to destination city from start city
TRACE OF TARGETS: ...                         [show me what these are – let it happen as normal]
# TARGETS:  ...                               [show me what this is – let it happen as normal]
#   #   #   #   #   #   #   #   #   #   #   #
Amsterdam (0) TO Dublin (19)
DISTANCE:  ?
PATH:  SORRY - can NOT get to destination city from start city
TRACE OF TARGETS: ...                         [show me what these are – let it happen as normal]
# TARGETS:  ...                               [show me what this is – let it happen as normal]
#   #   #   #   #   #   #   #   #   #   #   #
```

NOTES

1. *Data above is NOT correct – I’m just showing the formatting.*
2. ***USE THE EXACT FORMAT SHOWN ABOVE***
3. *FOR DEMO: Use landscape, Courier Font , size 8 font, smaller margins, 2-sided printing to print Log file (in WordPad) to save paper*
4. *The . . . parts would be filled in, of course*
5. *PATH must show cities from START-to-DESTINATION (and NOT show DESTINATION-to-START)*
6. *TRACE OF TARGETS*
   a. *startCity is NOT in the trace since it’s selected during initialization before the while loop*
   b. *wrap-around (seen above) happens during printing of Log file in WordPad -*
      the program code itself just writes a SINGLE LONG LINE of cities
   c. *the cities MUST appear in THE ORDER IN WHICH THEY WERE SELECTED.*
      Do NOT just write them out based on the INCLUDED array (which would show all selected target cities in city-number order instead – WHICH ISN’T HELPFUL!!!).

********************************* **NOTES** *********************************

- Adjacency matrix MUST be triangular, lower-left triangle, external, binary, shorts
- Row > Column for lower-left triangle
- Map data is EXTERNAL – it's never loaded into memory. When needed, cityName, cityCode or roadDistance is accessed from the external FILE.
- Use MUST use MY ALGORITHM IMPLEMENTATION for Dijkstra's MCP Algorithm
- map.getRoadDistance returns one of 3 possible kinds of values:
    - 0 for the diagonal (a virtual distance NOT actually IN the file
    - "infinity" (maxShort) for no-road-between-those-cities (from MapGraph file)
    - actual roadDistance in MapGraph file for the road between those 2 cities
- Trace of Targets
    - it's not part of the ALGORITHM – but add code to handle it
    - It print targets AS THE TARGET GET SELECTED – and NOT just a printout LATER, after the route's been found, of all the nodes that were INCLUDED
- Check your output answers (including path and trace of targets) for REASONABLENESS
    - What appears to be the shortest path looking at the physical map (though use caution for paths going near the Alps and other mountainous or lake-filled areas)
    - every city in PATH will be in the TRACE (except START)
    - many cities in TRACE may NOT be in the PATH (especially when DESTINATION is quite a distance from START)
    - the order of cities in TRACE will be in increasing distance from START
- 2 different COSTS are of interest to in a problem like this:

    1. the cost of the actual SOLUTION - i.e., the minimum cost path DISTANCE
    2. the cost of FINDING the solution - i.e., # TARGETS