CS3310 - Fall 2014
Kaminski

Asgn 6 – **Tour Michigan**
Dijkstra's Minimum Cost Path (Graph)

OVERVIEW

The program finds the shortest route between pairs of cities using Dijkstra's minimum cost path algorithm. Batch processing is used to facilitate testing, with the user's list of city pairs stored in a transaction file and the results going to a log file. The raw map data is in a file containing road distances between neighboring major cities in Michigan (and a few out-of-state cities just across the borders) using major roads only. But before any route-finding is done, Setup program converts the raw map data into a more efficient-to-use external adjacency matrix for use in the UserApp program.

PROJECT STRUCTURE:

- Setup and UserApp are 2 separate programs.
- Setup is a plain non-OOP utility program.
- Setup includes a prettyPrint method so the developer can check that the external binary file, MichiganMapData.bin, is correct.
- UserApp MUST be an OOP program (using MapData and Route classes).
- No UI class – UserApp's main does the transaction file handling and opening/closing of the Log file (passing log in to MapData and Route methods, as needed).
- No TestDriver program needed since you'll only be manually running 2 programs for the demo.

THE 4 DATA FILES

MichiganRawMapData.txt **INPUT TO SetupUtility PROGRAM**

[Ignore blank lines & comment-lines (beginning with %) – those are for the human reader.]
Two types of DATA lines:

1st) one long line with a list of all the cities in the UpperPeninsula ("the UP") in the form:

```
up([ crystalFalls, copperHarbor, . . ., stIgnace ]).
```

Assume that a city is in the LowerPeninsula ("the LP") if it's NOT in the UP.
HOWEVER, theBridge will be treated as being in BOTH the UP and LP in UserApp
(even though it's not listed in the UP list here).

2nd) one line per road (graph edge) in the form:

```
distance(kalamazoo, grandRapids, 51).
```

that is, cityA, cityB, roadDistanceBetweenAandB

NOTES:

- *all 2-word cities like Grand Rapids will be listed as one word, like grandRapids*
- *all cites start with small letters*
- *the data represents an UNDIRECTED graph, so a particular road is listed only ONCE in this raw data file, though it must be stored in the graph in both A,B and B,A.*

# # # # # # # # # # # # # # # # # #

MichiganMapData.bin **OUTPUT FROM SetupUtility PROGRAM**
INPUT TO MapData CLASS (in UserApp PROGRAM)

A BINARY file (with no field-separators, no <CR><LF>'s, numeric data stored as shorts).
Four types of data records:

1st) one headerRec containing n and upN, both shorts
2nd) the n by n square matrix for the graph edges, where each entry is a short
3rd) upCityNames list (there are upN of these) – all strings or charArrays
(with theBridge first, then all the UP cities from the RawData file's UP list
(IN THE ORDER that they appear in RawData's UP list))
4th) otherCityNames list (there are n of these) – all strings or charArrays
(all the cities NOT listed in #3 above)
(these are in the sequence in which they are encountered in the RawData's distance records (whether as cityA or cityB)

# # # # # # # # # # # # # # # # # #

CityPairsTestPlan.txt (TransData) **INPUT TO UserApp PROGRAM**

[Ignore blank lines & comment-lines (beginning with %) – those are for the human reader.]
Data lines' format: startCityName, a space, destinationCityName
NOTES:

- *all 2-word cities like Grand Rapids will be listed as one word, like grandRapids*
- *all cites start with small letters, as above*
- *there will be some invalid city names which must be handled appropriately, although NOT using hard-coded special cases.*

# # # # # # # # # # # # # # # # # #

Log.txt **OUTPUT FROM SetupUtility PROGRAM &**
FROM UserApp PROGRAM'S RouteFinder CLASS

NOTES:

- *the data values below are NOT necessarily CORRECT – what's below just demonstrates what the required format/wording/spacing/… must look like*
- *the ... parts shown in the trace and the route must be the actual list of city names*
- *ROUTE must display from START-to-DESTINATION (even though it would be easier to display DESTINATION-to-START)*
- **USE THIS EXACT FORMAT, including spacing ! ! !**

PRETTY PRINT (in Setup) produces the following

```
MichiganMapData.bin FILE DATA

N (all): 50 (includes cities 0 through 49)
N in UP: 15 (includes cities 0 through 14)
```

```
      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  ... 49
   ------------------------------------------------------------------
00 | 000 005 034 114 XXX XXX XXX XXX 004 XXX XXX XXX XXX XXX 010 ... XXX
01 | 005 000 XXX XXX XXX 057 XXX XXX XXX 023 XXX XXX XXX 098 XXX ... 056
02 | 034 XXX 000 XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX 057 ... 032
. . .
49 | XXX XXX XXX 078 034 XXX XXX XXX XXX XXX XXX XXX 047 XXX 093 ... XXX

ALL Cities:
00) theBridge
01) crystalFalls
02) copperHarbor
. . .
14) stIgnace,
15) michiganCity
16) southBend
. . .
48) claire
49) grayling
```

Route class method produces the following

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
START      :  paris
ERROR:  city not in Michigan Map Data
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
START      :  kalamazoo  (13)  Lower Peninsula
DESTINATION:  london
ERROR:  city not in Michigan Map Data
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
START      :  kalamazoo  (13)  Lower Peninsula
DESTINATION:  lansing  (11)  Lower Peninsula

TRACE OF TARGETS:  kalamazoo battleCreek southHaven grandRapids jackson
bentonharbor lansing (7 cities)

TOTAL DISTANCE:  82 miles
SHORTEST ROUTE:  kalamazoo > battleCreek > lansing
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
START      :  kalamazoo  (13)  Lower Peninsula
DESTINATION:  copperHarbor  (24)  Upper Peninsula
>>>>> 2 different peninsulas, so 2 partial routes shown <<<<<

TRACE OF TARGETS:  kalamazoo battleCreek grandRapids . . . theBridge (12
cities)
TOTAL DISTANCE:  320 miles
SHORTEST ROUTE:  kalamazoo > battleCreek > . . . > theBridge

TRACE OF TARGETS:  theBridge saultSteMarie . . . copperHarbor (6 cities)

TOTAL DISTANCE:  310 miles
SHORTEST ROUTE:  theBridge > . . . > copperHarbor
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

NOTES regarding the TRACE OF TARGETS above:

- *the wrap-around seen above happens during printing of the Log file in NotePad (or WordPad or…) - the program just keeps writing a single line to the file*
- *include the START city in the trace, even though it's not actually selected as a target inside the loop – (since it's "selected" first as part of the initialization before the big loop starts)*
- *cities must appear **IN THE ORDER IN WHICH THEY WERE SELECTED** – do **NOT** just write them out based on the Included array (which would result in city-number ordering)*

########################## **4 REQUIRED MODULES** ##########################

SetupUtility program

1. converts `MichiganRawMapData.txt` into `MichiganMapData.bin` including
   - creating a BINARY file which is RANDOM ACCESS (for the matrix part)
   - the upCityList STARTS WITH theBridge
     and then adds the cities in the up line in the RawData file
   - the otherCityList
     STARTS WITH michiganCity, then southBend (since those are the 1st cities encountered in the distance records in the RawData file which are NOT ALREADY IN the upCityList)
     and ENDING WITH (it looks like) claire and grayling since those are the last cities not yet in the upCityList, when processing the distance lines in the RawData file
   - EVERYONE will get the same exact cityNumber for a particular cityName because you MUST follow the above way of building the 2 lists
   - This is an undirected graph, so the matrix includes both A-to-B and B-to-A data
   - use 3 TYPES OF DATA VALUES in the matrix (as per class discussion) rather than 2 (as books/internet often show):
     i. maxShort's for the non-edges
     ii. 0's for the diagonal
     iii. actual edge values
   - SUGGESTION: This is just a utility, run just once. So you might use an internal 2D array and 2 internal ArrayLists (besides n & upN) to build the nicely structured data internally, before writing it all to the file
   - Use 1-pass loading (i.e., 1 pass through the RawData file) to load data into the 2D array and the 2 internal ArrayLists (with counters for n and upN) using the STREAM PROCESSING Algorithm - i.e., loop til EOF {read 1 line, deal with it}
2. prettyPrints the FILE once it's been created – displaying it on the Log file.
   [Don't just prettyPrint the internal temporary working structures since that isn't as useful as a debugging aid for the developer – you (and the grader needs to see if the FILE is correct].

# # # # # # # # # # # # # # # # # #

UserApp program - (the controller and transaction handler)

- declare a mapData & route objects
- process transaction file, one pair at a time
  - get a city pair,
  - find their city numbers (from mapData's getCityNumber),
  - find out which peninsula the cities are in (from mapData's getPeninsula)
  - if either city is theBridge, fix its peninsula to MATCH the other city's pen., since a MATCHED PAIR is better than a NON-MATCHED PAIR
  - write out intro info to Log
  - if UP/UP or LP/LP city-pair
    - find the shortest route (using Route's findShortestRoute),
  - else it's a UP/LP or a LP/UP city-pair, so for a more efficient search,
    1) find the shortest route from startCity to theBridge (using those city NUMBERS, not city NAMES)
    2) find the shortest route from theBridge to destinationCity (using those city NUMBERS, not city NAMES)
- finish up with mapData object (i.e., the file needs closing)

NOTE: Route's methods do the actual writing out the Trace of Targets, Total Distance and Shortest Route directly to the Log file, so that file will have to be passed in as a parameter.

# # # # # # # # # # # # # # # # #

MapData class (in a physically separate file)

Handles everything to do with MichiganMapData.bin file and MapData in general, including

- opening the file in the constructor
- loading the header data & city names into memory into n, upN, upCityNameList (an ArrayList) and cityNameList (an ArrayList)
  - BUT DO NOT LOAD THE MATRIX INTO MEMORY
  - [The cityNameList contains ALL THE CITIES, not just the LP cities for easier processing. So the first upN cities read are appended to BOTH the upCityNameList and the cityNameList.]
- closing the file in finshUp when requested by UserApp AT THE VERY END

Provides these public services:

- getCityNumber *(based on a cityName)*
- getPeninsula *(based on a cityName - assume LP if it's not in UP list)*
- getCityName *(based on a cityNumber)*
- getRoadDistance *(based on 2 cityNumber's)*
  - *WHICH GETS THE short FROM THE BINARY FILE USING RANDOM ACCESS*

NOTES:

1. *Do RANDOM ACCESS to the start of the cityNames to be able to load those into the 2 ArrayLists. Where in the MichiganMapData.bin file do the cityNames start? byteOffset calculation has to skip over the 2 shorts in the headerRec PLUS the n by n matrix of shorts.*
2. *EVERYONE will have constructed the city lists in the same way:*
   a. *theBridge*
   b. *the UP cities in the up list in the RawData file (IN THAT ORDER)*
   c. *the OTHER cities, AS THEY APPEAR in the distance records in the RawData file (with the start checked first, then then destination) IF THEY'RE NOT ALREADY IN THE LIST YET.*
3. *ArrayLists start at 0 not 1, so theBridge is cityNumber 0.*
4. *Neither cityName list is ever sorted.*
5. *getRoadDistance(cityA, cityB) uses RANDOM ACCCESS to read the correct short. To calculate byteOffset, allow for*
   a. *the sizeOfHeaderRec (2 shorts),*
   b. *the number of ROWS which came BEFORE cityA (a number) (and sizeOfARow is . . . the size of n shorts)*
   c. *the number of COLUMNS which came BEFORE cityB (a number) (and sizeOfAColumn is. . . the size of 1 short)*

# # # # # # # # # # # # # # # # #

Route class (in a physically separate file)

NOTE: This class needs access to Log file

NOTE: This class needs access to mapData methods like getRoadDistance and getCityName.

Storage: the 3 working array (all of size N, not MAX_N, since you now already know N):

distance, included, predecessor

Public services:

- findShortestRoute *(= the local controller which handles a Start/Destination pair of city **numbers. NOTE:** it is NOT this class's responsibility to handle city **names**)*

Local (private) methods used by findShortestRoute:

- initialize3Arrays
- searchForPath *(the "search" part of Dijkstra's Algorithm)*
  - *(PLUS it prints the TRACE OF TARGETS **as it's selecting them**)*
- reportAnswers *(the TotalDistance and the ShortestRoute)*
  - *(from Start to Destination, not Destination to Start)*

NOTES:

- The Start/Destination pair for this module are not always the cityPair file's Start and Destination cities. See note above for LP/UP and UP/LP mismatched pairs where theBridge is used as an intermediate city to make the search more efficient - in which case 2 partial routes are found and reported. But this is all controlled from UserApp – findShortestRoute has no idea this is happening.

- The TraceOfTargets and ShortestRoute specify cityNAME not cityNUMBER. So the methods above have to have access to MapData's getCityName service.

############################## **MY ALGORITHM** ##############################

You MUST USE THE ALGORITHM I SHOW IN CLASS for Dijkstra's Algorithm. Don't just take code/algorithm from a data structures book or the internet, (even though it would produce the same results) – BIG POINT-LOSS IF YOU DO.

My algorithm assumes that the graph contains THREE different possible values (not 2) in the graph, which differs from what's in the green booklet:

1) actual edge weights if there's an edge
2) 0 on the diagonal for a node to itself
3) "INFINITY" meaning there's NO edge (some books use 0's here too for "NO edge")