CS3310 – Spring 2013 **Asgn5 (Shortest Path)**
Kaminski **DrivingApp (for Europe)**

OVERVIEW: **DrivingApp** finds the shortest path on a road map for a designated start/ destination city pair. (A series of pairs are stored in a batch file rather than entered interactively). Dijkstra's Minimum Cost Path Algorithm is used (implemented based on the algorithm shown in class). The map is stored as a graph, implemented as an **INTERNAL ADJACENCY LIST**, with city names stored in a separate table to facilitate user interaction.

********** **PROGRAM** **********

DrivingApp (**MUST use OOP paradigm**) does the following:
Get fileNamePrefix from console user (YOU) - options include Europe and Other (for now)
Declare map, ui and shortestPath objects
Loop til ui indicates done-with-transactions
{ ask ui for start & destination cities (which will be city NAMES, not NUMBERS)
ask map for the 2 cities' numbers (given the above names)
ask shortestPath to FindPath (given the above 2 city numbers)
(which will find the answer and report it to ui for logging)
} *// may have to change this to a process-read loop structure instead, depending on. . .*
FinishUp with each of the 3 objects, as needed

********** 3 **CLASSES** (in physically separate files) **********

Map class

- handles all accessing of `?MapData.txt` file
- builds and stores the **internal graph** and the **cityName table**
- provides public services (methods) including:
  - GetCityName (given cityNumber)
  - GetCityNumber (given cityName)
    - Capitalize both target & name in table for search comparison
  - GetDistance (given cityNumber1 and cityNumber2)
    (i.e., FROM cityNumber1 TO cityNumber2)

NOTE: code outside this class has no idea how the graph or city name table is implemented & accessed. BUT INSIDE THIS CLASS, YOU MUST USE: ***Adjacency lists as explained in class***
[0 points for this part if you use an adjacency MATRIX]

ShortestPath class

- handles all path finding (i.e., Dijkstra's Algorithm) including reporting the trace and answers to ui's Log file

NOTE: code outside this class doesn't know how the shortest path answer is found - whether it's a search of a database of stored answers or a Google search for the answer or some algorithm for calculating it or. . .]
BUT INSIDE THIS CLASS, YOU MUST USE: ***Kaminski's pseudocode version*** *of Dijkstra's Shortest Path Algorithm - where you* ***ADD THE TRACE part****]*

- stores: 3 working-storage arrays,
- provides public services (methods) including:
  - FindShortestPath (given startNumber and destinationNumber)
- Includes private services (methods) including
  - Initialize (the 3 working-storage arrays)
  - Search
  - ReportAnswer (which includes both the PathDistance & the ActualPath)

UI class

- handles all accessing of `?CityPairs.txt` file and `Log.txt` file

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

`?MapData.txt`

1. neader line: N, a space, either U or D (for Undirected or Directed Graph)
2. N cityName lines (graph node names, corresponding to nodes 0 through N-1)
3. roadDistance lines (graph edges):
   A variable number of lines containing individual edge data in the form:
   cityNumberA space cityNumberB space distance (i.e., edgeWeight)
   [NOTE: For Undirected Graphs, ONE line describes both A-to-B and B-to-A
   For Directed Graphs, ONE line describes JUST A-to-B]

- *Node numbers start at 0 (not 1)*
- *cityNames area always single words (e.g., Grand Rapids would be GrandRapids)*
- *? is the fileNamePrefix supplied by the console user – either Europe or Other (for now)*

= =

`?CityPairs.txt`

Each line contains a pair of city NAMES: startCityName space destinationCityName

- cityNames area always single words (e.g., Grand Rapids would be GrandRapids)
- *? is the fileNamePrefix supplied by the console user – either Europe or Other (for now)*

= =

`Log.txt`

```
STATUS:  OK, Europe MapData loaded – contains 19 cities

For Developer testing purposes, here's the HeadPtr array
[0] 0
```

```
[1] 3
. . .                                          [. . . part filled in, of course]
[19] 42

% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
Amsterdam (0) TO London (X)
ERROR - one of these cities is not part of this map

% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
Kalamazoo (X) TO Amsterdam (0)
ERROR - one of these cities is not part of this map

% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
Paris (13) TO Copenhagen (6)

TRACE OF TARGETS:  Brussels Amsterdam Bern Genoa Hamburg Madrid Munich Rome
Berlin Trieste Copenhagen [11 targets]

DISTANCE:  907
SHORTEST ROUTE:  Paris > Brussels > Amsterdam > Hamburg > Copenhagen

% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
Warsaw (18) TO Warsaw (18)

TRACE OF TARGETS:  [0 targets]

DISTANCE:  0
SHORTEST ROUTE:  Warsaw

% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
```

<u>NOTES regarding Log file</u>

- the data above is not accurate – it is used just to show the proper format

- use the EXACT FORMAT shown above, including spacing

- The . . . parts would be filled in, of course

- SHORTEST ROUTE must show cities from START-to-DESTINATION
(even though it's slightly easier to show DEST.-to-START)

- TRACE OF TARGETS

- the wrap-around (seen above) happens during printing of Log file in WordPad
- the program just writes a single long line

- 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 the selected target cities in city-number order instead).

<u>OBSERVATIONS WHEN CHECKING FOR CORRECTNESS</u>

- every city in SHORTEST ROUTE definitely will be in the TRACE (except START)

- some/many cities in the TRACE will probably NOT be in the SHORTEST ROUTE
(especially when DESTINATION is quite a ways from START)

- the order of cities in TRACE will be in increasing distance from START

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

You MUST:

- use the program, class, object, method names described in the specs for easier readability/maintainability among different programmers
- put the program and 3 classes in physically separate files
- put top-comments on each of these code files
- use the OOP approach for the 3 classes - so store class-related variables as instance variables in the class (rather than passing a lot of parameters around) – and the methods inside the 3 classes are instantiable methods, not static methods – and you've appropriately specified methods as public or private
- have UI class contain all handling of CityPairs & Log files (open/read/write/close)
- use MY VERSION for Dijkstra's Shortest Path Algorithm
- use an INTERNAL Adjacency LISTS
- do Trace of Targets
    - Even though it's not part of the ALGORITHM handout
    - It print targets AS THE TARGETS GET SELECTED – and NOT just a printout LATER of all the nodes that were INCLUDED

Check your output answers (including the shortest route and trace of targets) for REASONABLENESS

- knowing what the shortest path would be (approximately)
- and what general pattern the Trace would follow

2 different COSTS are of interest:

1) the cost of the actual SOLUTION
    - i.e., the minimum cost path DISTANCE
2) the cost of FINDING the solution
    - i.e., # TARGETS as a measure of the number of times around the loop

The cost of FINDING the solution can be reduced substantially by using an AI approach – including the famous A* algorithm. But don't use A* for this asgn – use Dijkstra's more conservative algorithm.