CS3310 – Fall 2012 **Asgn5 (Shortest Path)**
Kaminski **DrivingApp (for Europe)**

OVERVIEW: **DrivingApp** finds the shortest path on a road map for designated start/ destination city pairs. (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 **EXTERNAL ADJACENCY MATRIX**. It is built by **SetupUtility** program from raw (linearly formatted) data. *[Like asgn 4, DrivingApp uses this graph as an EXTERNAL data structure – it does NOT load it into memory, other than the HeaderRecord].* This utility program also sets up a CityName file to facilitate user interaction. The program also "pretty-prints" the binary graph file and the CityName file to aid the developer during testing.

********** **PROGRAMS** **********

SetupUtility *(OOP optional - good modularization is required)*

Ask user (vial Console) which map data to use – options include (for now) just Europe
- that's used as the fileNamePrefix.

Create `?MapData.bin` file and `?CityNames.txt` file from ?`RawData.txt` file's data
(where ? is the fileNamePrefix)

1. create an INTERNAL adjacency matrix (2-dimensional array) & fill it with data
   You MUST USE MY APPROACH of 3 possible matrix values:
   - **"infinity" (implemented as int's maxValue) for non-edges**
   - **0's in diagonal**
   - **actual Edge values read in from the RawData file**
2. write out the MapData file and the CityNames file

Pretty-print `MapData` and `CityNames` files to `LogSession.txt` file
(opened in truncate mode)

= =

DrivingApp (**MUST use a proper OOP approach**) – the algorithm:

Declare 3 objects (use these names)**:**

- **map** – handles all accessing of MapData and CityNames files
- **ui** – handles all accessing of `?CityPairs.txt` file and `LogSession.txt` file
- **shortestPath** – handles all path finding (i.e., Dijkstra's Algorithm) including reporting the trace and answers to UI's LogSession file

Asks user (vial Console) which map data to use – options include (for now) just Europe
- that's used as the fileNamePrefix.

Loop til UI indicates Done

{ ask UI for the 2 cities (GetStartCity and GetDestinationCity)
(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. . .*

Call FinishUp methods for each of the 3 objects as needed

********** **CLASSES (USED BY DrivingApp)** **********

- **Map** – handles all accessing of MapData and CityNames files
  - N & CityNames are loaded into memory when the file is first opened.
  - RoadDistanceMatrix is NEVER LOADED INTO MEMORY – the distance values are always accessed directly from MapData FILE.
  - some public service methods needed:
    GetCityName(int cityNumber)
    For simplicity, at this time, just does linear search of the INTERNAL CityNames array
    GetCityNumber(string cityName)
    Does random access using the INTERNAL storage array
    GetRoadDistance(int cityNumber1, int cityNumber2)
    Does **RANDOM ACCESS using the EXTERNAL FILE**

  [NOTE: the outside world (i.e. DrivingApp) doesn't know how the graph is implemented/accessed inside here: internal or external, adjacency matrix or adjacency lists, structure of CityNames file, etc.]

- **UI** – handles all accessing of `?CityPairs.txt` file and `LogSession.txt` file

- **shortestPath** – handles all path finding (i.e., Dijkstra's Algorithm) including reporting the trace and answers to UI's LogSession file

  *[NOTE: outside world doesn't care how the shortest path answer is found, what algorithm is used, what working storage is needed, whether there's a DB search or algorithm for calculating i., 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,
    & trace array/list (STORES TARGETS **IN THE ORDER THAT THEY ARE SELECTED**)
  - public service method needed (others, as needed):
    - FindShortestPath (given startNumber and destinationNumber)
  - private service methods needed (others, as needed)
    - Initialize (the 3 working-storage arrays)
    - Search
    - ReportAnswer (which includes both the PathDistance & the ActualPath)
    - ReportTraceOfTargets (which includes both the Trace & # Targets)

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

<u>`?RawData.txt`</u> *(includes <CR><LF>'s)*

1. <u>Header</u> Line: N, a space, either U or D (for Undirected or Directed Graph)
2. <u>CityName</u> Lines (graph nodes):
   N variable-length lines of names for nodes 0 through N-1
   (NOT 1 THROUGH N)
3. <u>RoadDistance</u> 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)*

= =

<u>`?MapData.bin`</u> *(NO <CR><LF>'s, NO field-separators)* ***[This is a RANDOM ACCESS FILE]***

1. <u>Header Record</u>: N (an int)
2. <u>RoadDistance Matrix</u>: N x N (all int's) (i.e., 0's, "infinity's", actual edge values)

= =

<u>`?CityNames.txt`</u> *(NO <CR><LF>'s, NO field-separators)*

<u>CITY NAMES</u> strings which are **stored in FIXED-LENGTH LOCATIONS**

To determine the appropriate **location-length** (for every city name in the file), SetupUtilit determines the longest city name as it's reading the names from RawData file, and temporarily storing all the names in an INTERNAL array (of variable-length strings), before writing them to the file into with fixed-length locations. By the time they're being written to the file, SetupUtility will KNOW what the location-length IS, and so be able to write this with fixed-length locations – which means it'll need to seek to the start of each location before writing each one).

NOTE: When writing out strings, allow for the extra length field (for C# & Java) or the null-terminator (for C++ and C) in determining the location-length. Also, if your language uses Unicode(16-bit char's) rather than normal ASCII code (8 bit char's), take this into account in determining location-length.

This "fixed-length location" requirement allows it to be a RANDOM ACCESS FILE, although DrivingApp does NOT do random access AT THIS TIME because it's not actually using this file EXTERNALLY.

At this time, Map's constructor loads this file's strings into CityNames array in memory. And GetCityName and GetCityNumber access the INTERNAL storage.

IDEALLY (don't do this for A5), GetCityName and GetCityNumber would access CityNames file as EXTERNAL storage. So the file would have to be organized to do random access. But all such changes would be done in Map's constructor And GetCityName and GetCityNumber methods.

= =

<u>`?Trans.txt`</u> *(includes <CR><LF>'s) [EuropeTrans and OtherTrans versions]*

- Each line contains a pair of city NAMES: startCityName space destinationCityName
- cityNames area always single words (e.g., Grand Rapids would be GrandRapids)

= =

<u>`LogSession.txt` FROM SetupUtility</u> *[opened in TRUNCATE mode] (includes <CR><LF>'s)*

```
Map Data:  Europe   Number of cities:  19
Amsterdam
. . .
Warsaw
       0    1    2    3    4    5    6    7    8    9   10  . . .    18
   ------------------------------------------------------------------
 0|    0  512  123 1321  312  432  567 1111 1122  211 1321  . . .  1104
 1| 1133    0  444  449  512 1212  654  102  109 1001 1010  . . .   345
. . .
18|  123  234  345  456  567  678  789  890  111  222  333  . . .     0
```

<u>`LogSession.txt` FROM DrivingApp</u> *[opened in APPEND mode] (includes <CR><LF>'s)*

```
#   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #
Amsterdam (0) TO London (-1)
ERROR - one of the cities is not on this map

#   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #
Kalamazoo (-1) TO Amsterdam (0)
ERROR - one of the cities is not on this map

#   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #
Paris (13) TO Copenhagen (6)
DISTANCE:  907
PATH:  Paris > Brussels > Amsterdam > Hamburg > Copenhagen

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

#   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #
Amsterdam (0) TO Bern (3)
DISTANCE:  558
PATH:  Amsterdam > Bern

TRACE OF TARGETS:  Brussels Hamburg Paris Munich Bern
# TARGETS:  5

#   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #
Warsaw (18) TO Warsaw (18)
DISTANCE:  0
PATH:  Warsaw

TRACE OF TARGETS:
# TARGETS:  0
#   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #
```

NOTES regarding LogSessopm file

- USE THE EXACT FORMAT SHOWN ABOVE, including spacing
- Use Courier New Font & size 7 to print this out in WordPad to get a nice matrix without wraparound
- The . . . parts would be filled in, of course
- The data I've shown below is not necessarily correct (e.g., for the matrix or the PATH or TRACE of Targets – I'm just showing you the FORMAT
- PATH must show cities from START-to-DESTINATION (even though it's easier to show DEST.-to-START)
- regarding TRACE OF TARGETS
- the wrap-around (seen above) happens during printing of LogSession file in WordPad
- the program just writes a single long line – that's the idea
- 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).

OBSERVATIONS WHEN CHECKING FOR CORRECTNESS

- every city in PATH definitely will be in the TRACE (except START)
- many cities in TRACE may NOT be in the PATH (especially when DESTINATION is a ways from START)
- the order of cities in TRACE will be in increasing distance from START
- 2 different COSTS are of interest to us:
1) the cost of the actual SOLUTION - i.e., the minimum cost path DISTANCE
2) the cost of FINDING the solution - i.e., # TARGETS

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

You MUST:

- use the program, class, object, method names described in the specs for easier readability/maintainability among different programmers
- put the 2 programs and 3 classes in physically separate files
- put top-comments on each of these code files
- use OOP for DrivingApp program (though not necessarily for SetupUtility program)
- 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 house all handling of Trans & LogSession files (open/read/write/close)
- use MY ALGORITHM for Dijkstra's Shortest Path Algorithm
- use the "3 possible types of values" in the Adjacency Matrix (as described above and in class)
- use an EXTERNAL Adjacency Matrix (NOT an internal one, and NOT 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 path and trace of targets) for REASONABLENESS
    - knowing what the shortest path would be (approximately)
    - and what general pattern the Trace would follow