CS 213
Project – Geographic Information System

Given: January 18, 2011   **Due:** March 03, 2011

The assignment is due by 11:59 p.m. of the due date. The solution must be the student's own work. Assistance should only be sought or accepted from the course instructor. Any violation of this rule will be dealt with harshly. Follow the documentation guidelines while writing and documenting your programs. Programs that are not well-documented will be penalized heavily. Your program must work on *all* possible inputs.

Introduction

The input file `gis.dat` contains basic geographic data on 128 cities in North America. This project asks you to build a program that reads `gis.dat` and then efficiently responds to geographic queries about the data. Specifically, `gis.dat` contains, for each city, the following pieces of information:

- Name of the city

- State of the city

- Latitude and longitude of the city

- Population of the city

- Distances to all other cities

The population numbers are definitely out of date. The highway mileage may also be out of date, but the data serves our purposes quite nicely. Note that the data file specifies for each city, the distances to all cities before it in the file; this is sufficient to get pairwise distance between any pair of cities. Also note that latitudes and longitudes are specified without decimal points or N, S, E, W indicators: for example Youngstown, OH is at latitude 41.09972 N and longitude 80.64972 W and this is specified as "[4110,8065]." Starting with this data, your program should, eventually, be able to answer queries such as:

1. How many cities with at least 100,000 people are there between latitudes 40 N and 50 N and longitudes 85 W and 100 W (i.e., the upper midwest)?

2. Is it possible to get from Yakima, WA to Wilmington, DE in 10 hops, where each hop connects a pair of cities that are at most 200 miles apart?

3. What is the estimated length of the shortest tour that visits each city in the northeast (roughly, 65-80 W, 38-50 N) exactly once?

Your program will end up using the Graph classes from *NetworkX* (`http://networkx.lanl.gov/`), a fair bit, in order to answer queries such as those mentioned above.

The Gis class

Start by designing and implementing a class called `Gis` that is reponsible for reading `gis.dat`, storing all of the information in appropriate data structures, and then answering queries efficiently.

The `Gis` class must support the following methods.

1. `selectCities(attribute, lowerBound, upperBound)`: This method will be used to "select" a set of cities that satisfy some conditions. For example, this method can be used to select all cities in the upper midwest that have population 100,000 or more. The argument `attribute` is a string that can be one of "name", "state", "latitude", "longitude", and "population." Depending on the value of `attribute`, the arguments `lowerBound` and `upperBound` specify a range. So for example, if attribute is "population" then setting `lowerBound` to 100,000 and `upperBound` to 500,000 will select all cities whose population is between one hundred thousand and five hundred thousand (inclusive). Here is another example: if attribute is "name" then setting `lowerBound` to "R" and `upperBound` to "T" will select all cities whose names are lexicographically in the range between "R" and "T" (inclusive). These examples should indicate to you that while `attribute` is guaranteed to be a string, `lowerBound` and `upperBound` will have different types, depending on the value of `attribute`.

   An important property that `selectCities` is required to have is that it selects only from those cities that are already selected. This allows us to use `selectCities` repeatedly to select a set of cities that satisfy several constraints. For example, we could first call `selectCities(``population'', 100000, 500000)` to select cities whose population is in the range 100,000 to 500,000. We could then call `selectCities(``name'', ``R'', ``T'')`. This latter call would select cities with names between "R" and "T" from only among those cities already selected, i.e., those that have population in the range 100,000 to 500,000. As a result, the set of cities selected at this point is all cities with names between "R" and "T," whose population is in the range 100,000 to 500,000. You should be able to see that by calling selectCities repeatedly, one can select a set of cities that satisfy several constraints, e.g., all cities in California whose names start with "S" and which have population more than 200,000.

2. `selectAllCities()` and `unselectAllCities()`: These methods respectively select and unselect all cities.

3. `selectEdges(lowerBound, upperBound)`: Here `lowerBound` and `upperBound` specify a "distance range." For example, if `lowerBound` is set to 0 and `upperBound` is set to 500, this method will select all edges between pairs of cities whose distance (as specified in `gis.dat`) is at most 500 miles. For now the only criteria we will use to select edges is this distance criteria. Assume that initially no edges are selected.

3. `selectAllEdges()` and `unselectAllEdges()`: These methods respectively select all edges and unselect all edges.

4. `makeGraph()`: This method makes and returns a graph whose vertex set is the set of selected cities and whose edge set is all selected edges connecting pairs of selected

cities. For example, suppose you have selected all cities with population 100,000 or more and all edges between pairs of cities whose pairwise distance is at most 200 miles. Then `makeGraph()` will construct and return a graph whose vertex set is the set of all cities with population 100,000 or more and with edges connecting pairs of such "high population" cities that are at most 200 miles apart. Using algorithms studied in class, we can find out if in such a graph it is possible to travel from any "high population" city to any other "high population" city, without visiting any "low population" city and furthermore using only hops of length at most 200 miles.

5. `printCities(attribute, choice)`: As before, `attribute` can be one of "name", "state", "latitude", "longitude", and "population." This methods prints all selected cities sorted in increasing order by the given attribute. For example, if the attribute is "latitude" then the selected cities will be printed in increasing order of latitude (i.e., south to north). You should also implement `printCities()`, i.e., with no given attribute, which should behave exactly like `printCities(``names'')`. The second parameter `choice` ('S' or 'F') determines whether the requested output should be displayed in "short" form or "full" form.

6. `printEdges()`: This should print all selected edges, in no particular order.

You will also be adding several other methods based on the driver program given below. In particular, let's discuss the following three methods.

- `testMinMaxConsDistance()`: This function starts by querying the user for a `source` and `target`. It then minimizes the maximum distance between consecutive cities on a path between the source and the target on the network induced by the selected cities and selected edges. It also outputs the actual path between the source and the target using an edge with that cost. It keeps querying the user for `source` and `target` until the user enters nothing for the source or for the target, i.e., just enters <Enter>.

- `tour(start)`: This method outputs a traveling salesman tour on the selected cities (and using the selected edges) starting from `start`. The tour is computed using the nearest neighbor heuristic, i.e., at any point in the tour, the next city chosen is the closest to to current city among all the unvisited cities. In the output, exactly four cities are printed on each line except possibly the last line of the output which may have fewer than four cities.

- `minCut()`: This method outputs the minimum cut in the graph induced by selected cities and selected edges. You must implement the Stoer-Wagner algorithm to find the min-cut. You may find the `heapq` module useful.

Feel free to add any other functionality that you may feel appropriate. Place yourself in the role of a user of the `Gis` class and ask yourself what other methods might be useful. Also, if you notice the same code fragment repeating in several methods, you should probably turn that into a "helper" function that can be called by other methods.

Your class must be designed based on the following client program.

```python
import gis

def main():
    gsystem = Gis()

    gsystem.selectAllCities()
    gsystem.selectAllEdges()

    delimiter = '\n**************************************\n'


    #### EXPERIMENT 1 ####
    gsystem.printCities()
    print delimiter

    # print full display
    gsystem.printCities('population', 'F')
    print delimiter

    #### EXPERIMENT 2 ####

    # select all cities with latitudes between 40N and 50N
    # and longitudes between 85W and 130W.
    gsystem.selectCities('latitude',40,50)
    gsystem.selectCities('longitude',85,130)

    print 'Population distribution of cities with latitudes between\n
    40N and 50N and longitudes between 85W and 130W.\n'

    gsystem.printPopulationDistr()

    print delimiter

    # print population distribution of cities in CA
    gsystem.selectAllCities()
    gsystem.selectCities('state','CA')

    print 'Population distribution of cities in California.\n'
    gsystem.printPopulationDistr(30000)

    print delimiter

    #### EXPERIMENT 3 ####

    # print 'num' most populated states in non-increasing
```

```
    # order of their population.

    gsystem.selectAllCities()

    num = 3
    gsystem.printPopulatedStates(num)
    print delimiter

    #### EXPERIMENT 4 ####
    gsystem.testMinMaxConsDistance()
    print delimiter

    #### EXPERIMENT 5 ####
    gsystem.selectAllCities()
    gsystem.selectAllEdges()

    # print TSP tour starting from Yakima, WA, with
    # exactly 4 cities on each line except possibly the
    # last line.
    gsystem.tour('Yakima, WA')

    print delimiter

    gsystem.unselectAllEdges()
    gsystem.tour('Yakima, WA')
    print delimiter

    #### EXPERIMENT 6 ####
    gsystem.selectAllCities()
    gsystem.selectAllEdges()
    gsystem.selectEdges(1500,3000)

    gsystem.minCut()
    print delimiter
```

Suppose `gis.dat` contains the following data.

```
Youngstown, OH[4110,8065]115436
Yankton, SD[4288,9739]12011
966
Yakima, WA[4660,12051]49826
1513 2410
Worcester, MA[4227,7180]161799
2964 1520 604
```

When the client program is run on the above input we get the following output.

```
Worcester, MA
Yakima, WA
Yankton, SD
Youngstown, OH


**********************************
Yankton, SD [4288, 9739], 12011
Yakima, WA [4660, 12051], 49826
Youngstown, OH [4110, 8065], 115436
Worcester, MA [4227, 7180], 161799


**********************************

Population distribution of cities with latitudes between
40N and 50N and longitudes between 85W and 130W.

[0, 20000] :  1
[40000, 60000] :  1


**********************************

Population distribution of cities in California.

No cities found



**********************************

3 most populated states.
-------------------------------------------
MA 161799
OH 115436
WA 49826


**********************************

Goal:  minimize the maximum distance between any pair of
consecutive cities on path from source to destination.

Source (City, State):  Yankton, SD
Target (City, State):  Worcester, MA

Cost of optimal solution:  966
```

```
Path from Yankton, SD to Worcester, MA:
Yankton, SD
Youngstown, OH
Worcester, MA


***********************************

Source (City, State):
Target (City, State):
***********************************

Traveling Salesman Tour starting from Yakima, WA is as follows.

Yakima, WA -- > Yankton, SD -- > Youngstown, OH -- > Worcester, MA -- >
Yakima, WA

Tour length:  6047
***********************************
Traveling Salesman Tour starting from Yakima, WA is not possible.


***********************************

The edges in a min-cut are as follows.

(Yakima, WA , Youngstown, OH)

Weight of the min-cut:  2410


***********************************
```

Discussion

This project gives you a glimpse into how a geographic information system (GIS) might work. At the back end of the system there are multiple sources of geographic data (in our case, all our data comes from `gis.dat`). At the front end we have a "query engine" that efficiently responds to geographic queries from the user. The user of a GIS, typically has no knowledge of the data sources, size and format of the data files, etc. that the back end is dealing with. In our context, this separation translates into the testing programs, like the above driver that has no knowledge of or access to `gis.dat`. Visualization of data is a critical part of most GIS software - unfortunately, we will not be incorporating it into this project. However, if interested, you should be able to use the graph visualization component of NetworkX to do the visualization.

Submission Guidelines.   You must create a directory called P_<last name> which must contain only the files that are needed for your program.

Before you submit your program, you must create a *compressed*, *tar* file of the directory P_<last name>. This can be done in two steps as follows.
```
% tar cvf P_<last name>.tar P_<last name>
% gzip P_<last name>.tar
```
The above commands should create a file P_<last name>.tar.gz . You must e-mail the above file as an attachment to `rajivg@crab.rutgers.edu`. The programs are due by 11:59p.m. of the due date. Hard copies of your programs must be turned in by the end of the first class after the deadline.

Some other commands that may be useful are the following.
Uncompressing a file can be done using the `gunzip` command.
```
% gunzip <file name>.gz
```
Extracting files from an archive can be done as follows.
```
% tar xvf <file name>.tar
```
To list all the files in the archive do the following.
```
% tar tf <file name>.tar
```