Introduction

Locating "facilities" in a geographic area so as to minimize cost and maximize coverage is a problem that lot of companies and organizations solve all the time. For example, Walmart has one of the largest private distribution networks in the world. Their webpages tell us that they have 40 regional distribution centers with each distribution center supporting "between 75 and 100 stores within a 250-mile radius." Because the distribution centers are so huge (over 1 million square feet in size) and cost so much to build and maintain, Walmart has to think very carefully about where to locate each distribution center. Similarly, a company such as Google has a bunch of "data centers" across the world, where its servers are located. Google also has to think very carefully about where to locate its data centers so that it can continue to provide high speed access to its search engine and other services. Even local governments have to solve the facility location problem when determining where to build the next public school or where to build a public park.

In this project you will be provided some basic geographic data on 128 cities in the U.S. and asked to solve a version of the *facility location* problem. Roughly speaking, your program will do three things: (i) read the provided geographic data from a file and store the data in an appropriate *data structure*, (ii) solve the facility location problem on this geographic data set, and (iii) output the solution to the facility location problem in a format that can be visualized in a mapping software such as *Google maps*.

Data

We have posted a file called `miles.dat` that contains geographic data on 128 cities in the U.S. and Canada. The data is very old — from a 1949 Rand McNally mileage guide, so the population numbers are definitely out of date. The highway mileage may also be out of date, but the data serves our purposes quite nicely. For each of the 128 cities, the data file contains (i) the name and state of the city, (ii) the latitude and longitude of the city, (iii) the population of the city, and (iv) the distance from that city to each of the cities listed before it in the data files. For example, Wilmington, Delaware is the tenth city in the data file and here is how information about Wilmington, DE appears in the data file.

```
Wilmington, DE[3975,7555]70195
466 168 1618 430 934 299 2749 1305 345
```

The two numbers in square brackets after "DE" are the latitude and longitude of the city. Following this, we see the population of the city, which was 70,195 in 1949. In the next line are the distances between Wilmington, DE and the 9 cities that appear before it in the file. For example, 466 is the number of miles between Wilmington, DE and the city Wilmington, NC that appears just before Wilmington, DE in the data file. Similarly, 168 is the distance between Wilmington, DE and Winchester, VA which appears just before Wilmington, NC in the data file.

The facility location problem

There are many versions of the *facility location* problem. Here is the version you are required to solve. You are given a *radius of coverage* r (in miles) and asked to locate the fewest number of facilities (each facility being located at one of the 128 cities in the data set) so that every one of the 128 cities in the data set is within r miles from some facility. For example, imagine that you are the logistics manager for a new retail chain and you want to locate distribution centers in the U.S. Based on your research you know that you should ensure that every population center is within 400 miles of at least one of your new distribution centers. In other words, each distribution center has a radius of coverage of 400 miles. Now given this constraint, you want to keep your costs low by building the fewest distribution centers.

No one knows how to solve the facility location problem efficiently. It is one of these notorious NP-complete problems that have eluded efficient solution for many decades now. In fact, there is consensus among computer scientists that problems like facility location will not have an efficient solution. However, companies still need solve the facility location problem all the time and so we will resort to using a "heuristic" that is not guaranteed to produce a solution with fewest facilities, but will hopefully find a solution that is pretty good.

The algorithm we want you to implement is described in the following pseudocode. The algorithm picks locations of facilities "greedily" by repeatedly finding a city that can "serve" the most number of cities that have not yet been "served." Since r is the given radius of coverage, a facility at a city c can "serve" any other city that is within r miles of c.

Greedy Algorithm for facility location

1. Initially all cities are `unserved`.
2. **while** there are cities that are `unserved`:
3. Pick a city c that "serves" the most `unserved` cities.
4. Mark the city c and all cities within r miles of c as `served`

Organizing your work

We require that you organize your work into three phases. Phase 1 is due on Friday, March 23rd, Phase 2 is due on Wednesday, March 28th, and the complete project is due on Wednesday, April 4th.

Phase 1

For Phase 1 you should write a program that reads the data in the given file `miles.dat` and stores this data in an appropriate data structure. Now that you have some experience in programming with lists in Python, there are many different possible data structures you could use. Here is our suggestion on how you should store the data. Define a list of strings called `cities` for storing the names of the cities. Since the names of cities are not all distinct (e.g., there are two Wilmington's) you should store the name of the city along with the name of the state that the city belongs to. One way to do this would be to represent each city by a string "cityName stateName". For example, the first city in the data set is Youngstown, which is a city in Ohio. You would represent this city by the string "Youngstown OH" with the city name and the two-letter state code being separated by a singe blank. The cities should by stored in the list `cities` in the order in which they appear in the data file. For example, `cities[0]` should be "Youngstown OH", `cities[1]` should be "Yankton SD" etc. Besides the list `cities`, you should maintain three other lists: `coordinates`, `population`, and `distances` for the rest of the information. All of these lists, including `cities`, should have length 128. We will now describe each of these lists.

- The element in position i in the list `coordinates` should be a list of length 2 specifying the latitude and longitude of the city in position i in the list `cities`. For example, `coordinates[0]` should be `[4110, 8065]`, the latitude and longitude of Youngstown, OH.

- The element in position i in the list `population` should be an integer specifying the population of the city in position i in the list `cities`. For example, `population[0]` should be `115436`, the population of Youngstown, OH.

- The element in position i in the list `distances` should be a list of length 128 specifying the distances between the city in position i in the list `cities` and the rest of the cities. For example, `distances[0]` should contain the distances between Youngstown, OH and the rest of the cities in the data set. In particular, `distances[0][0]` should be the distance between Youngstown, OH and itself, which is 0; `distances[0][1]` should be the distance between Youngstown, OH and Yankton, SD, which is 966; `distances[0][2]` should be the distance between Youngstown, OH and Yakima, WA, which is 2410, etc.

Together, the four lists `cities`, `coordinates`, `population`, and `distances` contain all of the data that is in `miles.dat`.

To complete Phase 1, you are required to implement the following functions:

- A function called `getCoordinates` that takes a city name and returns the list containing the latitude and longitude of the given city. You can assume that the city name is a string of the form "cityName stateName".

- A function called `getPopulation` that takes a city name and returns the population of the given city. You can assume that the city name is a string of the form "cityName stateName".

- A function called `getDistance` that takes two city names and returns the distance between the two cities, You can assume that the city names are strings of the form "cityName stateName".

- A function called `nearbyCities` that takes a city name c and a positive real number r and returns the list of all cities with r miles of the city c. You can assume that the city name c is a string of the form "cityname statename". The list of cities returned by your function should be in alphabetical order.

You should feel free to implement as many other functions as you think would be helpful to you, but you are *required* to implement the functions mentioned above and these are the functions we will create test files for.

You should submit a file called `project1Phase1.py` consisting of your solution to Phase 1 of Project 1.

Phase 2

In this phase you will write a function called `locateFacilities` that takes as parameters the data structure you built in Phase 1 and a positive real number r (representing the radius of coverage) and returns a list consisting of cities at which facilities were located. More specifically, you should use the following function header:

```
def locateFacilities(cities, distances, r)
```

The list of cities returned by this function should be in alphabetical order.

This function should implement the greedy algorithm described in the section titled "The facility location problem." One data structure that would be helpful in implementing this greedy algorithm is a boolean list called `served`. This is a list of length 128 that is initialized to all `False`

values. The element in position i in this list indicates whether city i (i.e., the city in position i in the list `cities`) has been served or not. Once such a data structure has been defined, you can then find out for any given city c the number of cities, *not yet served*, that are within radius r of c (you should write a function for this!). If you implement this, the greedy algorithm is easy to complete because it repeatedly locates a facility at a city c for which the number of "unserved" cites within radius r of c is maximum.

You should submit a file called `project1Phase2.py` consisting of your solution to Phase 2 of Project 1.

Phase 3

In this phase you will write a function called `display` that takes a list of cities and the data structure you built in Phase 1 and produces a file that can then be visualized using a mapping software such as Google Earth or Google maps. Suppose that L is a list of cities representing some subset of the 128 that are in the data set. At the minimum your visualization should consist of (i) a map of the U.S., with the cities in L identified by "balloons" or "push pins," (ii) every city in the data set that is not in L should be "connected" by a line segment to the nearest city in L. If L is the set of cities computed by your facility location program, then this visualization is a simple way of showing L and how the rest of the cities (i.e., those not in L) are served by the facilities in L. You can be as creative with this as you want and go well beyond these requirements. You may receive a maximum of 10 points of **extra credit** for the creativity, elegance, etc. of your visualization.

My *suggestion* for the function header for the function `display` is:

```
def display(facilities, cities, distances, coordinates)
```

This function takes the cities in the list `facilities` and places push pins at these cities. For the remaining cities, line segments are drawn from each of these to the nearest city in `facilities`. The data structures `distances` and `coordinates` created in Phase 1 will be helpful in figuring out (i) which facility is nearest and (ii) the latitude and longitude of each city for the KML file.

The visualization can be created by having your python program output what is called a *KML* file. "KML" stands for "keyhole markup language," which is a file format similar to html, except that it is used to display *geographic data* in an Earth browser such as Google Earth, Google Maps, or Bing maps. You can learn the basic rules of KML by reading KML Tutorials available on the web — a link to one of these is posted on the course page. Also, you can experiment with the sample KML files posted on the course page and those available at the KML tutorial page. There are two ways of viewing KML files:

- You can download Google Earth on to your home computer and open a KML file using Google Earth.
- You can place the KML file in a special directory on your Computer Science account. Anything placed in this special directory is "served" by the Computer Science web server and can be viewed by anyone using Google maps or Bing maps view their browser.

More specific instructions on viewing KML files are posted on the course page. KML files can be created "by hand" by editing a text file. In this instance, it is your program that needs to create a KML file.

Submit a python file called `project1Phase3.py` along with two KML files: `visualization300.kml` and `visualization800.kml`. The facilities shown in `visualization300.kml` are facilities returned by the `locateFacilities` function with $r = 300$. The facilities shown in `visualization800.kml` are facilities returned by the `locateFacilities` function with $r = 800$.
