HW 5: City Navigator (or USA*)

I have written a program to load in a geographic file and make queries on it. Your job is to create the `AStarGraph` that it relies on, in order to find the shortest path from one point to another. In addition, it can give information on single cities. Here is an example of running the program on a file containing the US state capitals:

```
$ java CityNavigator US-capitals.geo
File "US-capitals.geo" has been loaded with 50 cities.
Please enter your query:

> Olympia-Salem
Path found: Olympia - Salem (257 km)

> Olympia-Phoenix
Path found: Olympia - Salem - Carson City - Phoenix (2248 km)

> Pierre-Atlanta
Path found: Pierre - Lincoln - Jefferson City - Nashville - Atlanta
(2297 km)

> Austin-Boston
Path found: Austin - Little Rock - Nashville - Frankfort - Columbus -
Harrisburg - Albany - Boston (3304 km)

> Olympia-Tallahassee
Path found: Olympia - Boise - Cheyenne - Lincoln - Jefferson City -
Nashville - Montgomery - Tallahassee (4804 km)

> Honolulu
Honolulu is located at (21.3° N, 157.82° W). It is not connected to any
other city.

> Honolulu-Sacramento
I'm sorry. There's not a path from Honolulu to Sacramento.

> !exit
Goodbye!
```

Est. 1888 UNIVERSITY of PUGET SOUND

The `.geo` file contains the cities between which to travel and their latitudes and longitudes, followed by the distances of roads between them. Here is an example of a small file:

```
Seattle    47.6061    -122.3328
Tacoma     47.2529    -122.4443

Seattle    Tacoma     54
```

A negative number means the city is in the western or southern hemisphere. All distances are in kilometers.

Your `AStarGraph` must contain the following public methods:

- `AStarGraph()` (constructor) (O(1)). Creates an empty graph.
- `void addCity(String name, double latitude, double longitude)` (amortized O(1)) Inserts a city into the graph. Throws an `IllegalArgumentException` if a city by that name already exists. Note that city names often contain spaces.
- `void addRoad(String city1, String city2, double length)` (amortized O(1)) Adds a new two-way road between two cities. It should throw an `IllegalArgument Exception` if the cities don't exist, if the road is shorter than the shortest distance between the cities, or if the cities are already connected by a road.
- `boolean deleteRoad(String city1, String city2)` (amortized O(1)) Removes a road between two cities. Returns `true` if successful, or `false` if there was no road. Throws an `IllegalArgumentException` if the vertices don't exist.
- `String[] findPath(String city1, String city2)` ($O(n \log n)$) Uses A* to find the best path between two cities. It should throw an `IllegalArgumentException` if the cities don't exist. The return value should be an array of strings where `city1` is the first element, `city2` is the last element, and the rest of the cities indicate the step-by-step path between them. It will return `null` if there exists no such path.
- `double measurePath(String[] path)` ($O(p)$ where p is the path length) Sum up the length of the path given. It should throw an `IllegalArgumentException` if two cities adjacent in the list do not share a road.
- `int size()` (O(1)) Return the number of cities in the graph.
- `boolean isValidCity(String city)` (O(1)) Returns `true` if the city exists, or `false` otherwise.
- `double[] getCityLocation(String city)` (O(1)) Returns the location of a city, or `null` if no such city exists.
- `double getRoadLength(String city1, String city2)` (O(1)) Returns the length of a direct road between the cities, or `-1` if there is no such road.
- `String[] getNeighboringCities(String city)` (O(1)) Gets a list of all adjacent cities to the given one. It may return an array of length 0 if the city is isolated.

It is assumed that the graph is sparse—the maximum degree of a city is a small number. As such, a method may loop through the adjacency list of one or two cities, and still be considered to be constant time.

Note that you do not ever have to delete a city. And of course, style matters (and modifying the above appropriately if you are not using Java).

The distance heuristic to use for A* should be the "crow flies" distance, assuming the earth to be a perfect sphere. To find this distance for two locations A and B, use the following formula derived from the law of cosines, using their latitudes and longitudes:

$$d = \arccos\left(\sin(lat_A) \times \sin(lat_B) + \cos(lat_A) \times \cos(lat_B) \times \cos(lon_A - lon_B)\right) \times R$$

Here, R is the radius of the earth: 6371 km. Keep in mind the conversion between degrees and radians that you may need to do.