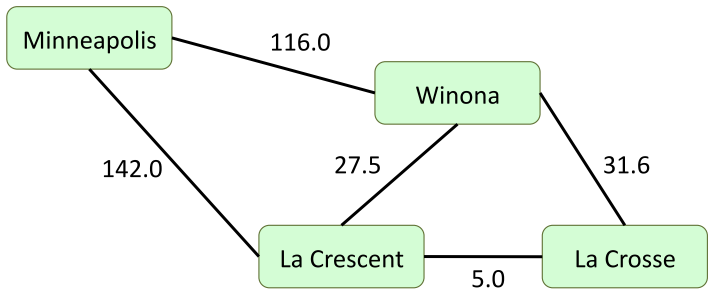
Artificial Intelligence (CS 452/552)
Assignment 01 (50 points)

Due before class, Wednesday, 27 September 2017



Your goal for this assignment is to write a program that reads in an input file; this file will describe a set of cities, and your code will then allow the user to search for shortest paths between them using A^* search. A city navigation file will have the following format:

```
# name latitude longitude
<CITY NAME 1> <LAT 1> <LON 1>
  ...
<CITY NAME N> <LAT N><LON N>
# distances
<CITY NAME X1>, <CITY NAME Y1>: <MILES BETWEEN CITY X1 AND CITY Y1>
  ...
<CITY NAME XM>, <CITY NAME YM>: <MILES BETWEEN CITY XM AND CITY YM>
```

That is, the file describes a set of N cities, with fixed (latitude, longitude) locations, defined on the first $N + 1$ lines of the file.[1] The next set of lines describes distances between pairs of named cities (lines starting with '#' are comments). Each pair of cities appears at most once in this list. If some pair of cities is in the list, then the distance between them is the same in both directions. If some pair of cities is not in the list, then there is no direct connection between those two cities, and any path between those cities must pass through some other city first (if any such path exists).

As an example, consider the small sample file included with this document (`cities01.txt`):

```
# name latitude longitude
La Crosse 43.8 -91.24
La Crescent 43.83 -91.3
Winona 44.06 -91.67
Minneapolis 44.98 -93.27
# distances
La Crosse, La Crescent: 5.0
La Crosse, Winona: 31.6
La Crescent, Winona: 27.5
La Crescent, Minneapolis: 142.0
Winona, Minneapolis: 116.0
```

[1] Data modified from: `http://www.cs.columbia.edu/~bert/courses/3137/`.

This file contains 4 cities, and their names and locations are given on lines 2–5. After that, we have the inter-city distances. For example, the distance between La Crosse and La Crescent is 5 miles, in either direction. From La Crosse, we can get directly to La Crescent or Winona; however, there is not distance given between La Crosse and Minneapolis, meaning that any path between those cities must pass through one of the others first.

For full points, your program will work as follows:

1. (*5 pts.*) It will run from the command-line, taking a string as a command line parameter. That string will be the name of a text file. For example, if you write a main Java class named `Search`, then we would call your program as follows:

   ```
   java Search cities01.txt
   ```

   If your code does not work this way, then it can cost you points on the final grade. In particular, your code should not require that we edit the code and re-compile each time we need to change input files, and it shouldn't ask the user what file to use after it starts running. When the code runs, it should open the file given and load the associated data for city names, locations, and distances.

2. (*5 pts.*) Once the file is processed, the code should prompt the user for input (using standard IO channels, e.g. `System.in` and `System.out` in Java). It should ask for the names of a starting city, S; if that city is not included in the input file, then it should explain to the user, and ask for another city, repeating until the user enters a valid city name. If the user enters the number `0` instead of a city name, the program should terminate. If the user enters a valid name for starting city S, then the program should ask for the name of a goal city G. Again, it should repeat the question if the name given is not valid, and terminate on input `0`.

3. (*30 pts.*) Your program will then perform A^* search in order to find the shortest path from start city S to goal city G, if such a path exists. When doing A^* for any city C, the cost function $g(C)$ is simply the total actual distance between start city S and C in the search so far. The heuristic value $h(C)$ is given by the length of the shortest possible arc between C and goal G on the surface of the Earth. This is guaranteed to be an admissible and consistent heuristic for the data sets given, if calculated correctly. Details of how to calculate this distance, using the **Haversine formula** can be found at:

   `http://andrew.hedges.name/experiments/haversine/`

   Note: for this to work, the inputs to any of the trigonometry functions must be in the form of *radians*, *not degrees*. Since the data given in the input file has latitude and longitude in degrees, you will need to convert. Also, since distances in the input are given in miles, you should also use miles for the radius of the Earth.

4. (*5 pts*) When the $A^\star$ search is complete, the program will print out the final path that it finds from the start point to the goal, or `NO PATH`, if no such path exists. Following that, there should be a record of:

   a. The total distance along that path (if no path exists this should be `-1`).

   b. How many nodes it generated in the search (i.e., nodes ever placed into the frontier).

   c. How many nodes are left in the frontier (i.e. nodes that are never expanded).

In the above example, for instance, if the user chooses to search for a path from La Crosse to Minneapolis, the output might look like this:

```
Route found:  La Crosse -> La Crescent -> Minneapolis
Distance:  147.0 miles

Total nodes generated:  9
Left in open list:  3
```

Note: while your code should give the same path and distance shown here (as this is the optimal solution), the number of nodes generated *may* be different. Some of this can depend upon exact implementation. If you see numbers that are far different from what is seen here, there may be a problem, however.

Another note: for full points, your code should be efficient enough that it can handle both the small sample file (`cities01.txt`) and the larger one (`cities02.txt`), which contains data for several hundred cities. An implementation that does not scale will lose 5 points.

5. (*5 pts.*) As explained in class, the most basic specification of $A^\star$ allows a single city to repeat multiple times during a search path. (**Note**: the algorithm will not output these non-optimal paths as solutions, but it will still explore them as it tries to find the optimal path.) For domains with non-decreasing path costs (like this one), this is inefficient and we will want to avoid it. For full points, your first implementation should leave this inefficiency in, so repeated nodes are allowed during search. Once that is complete, make a second version of the program, where the inefficiency is removed, and any repeated nodes are ignored along a single path.

You will hand in source-code for the program. If you complete the assignment, and have two versions of the algorithm, you will hand in two sets of source files (put each version in a separate folder for ease of organization). The instructor will compile your code and test it by giving it different input files. Two sample files are provided, and you can create your own for testing, but you need to use the same format as given above. Submit the work by uploading a compressed folder containing all of your source to D2L by the due-date and time (before class on the date given at the start of the assignment).

I prefer that you code the solution in Java, but if you wish to use another language, let me know first. When doing your coding, follow these conventions:

1. If you are not using Java, provide a short `README` file with your code, explaining exactly what commands or procedures are needed to make it compile and run properly. (If you use Java, and the program works from the command line exactly as described, this isn't necessary.)

2. Follow basic software engineering principles; that is, your code should be clean and well-structured, with comments explaining each method or function, and the usual format conventions to make it readable.