CSCI-331 Programming Project 01

Route Finding Using Tree Searches

Due: Friday, February 18th, 2022 by 11:59pm
Late Projects will be accepted up to 24 hours late and will be subject to a 20% penalty.

Project Objective

The objective of this project is to find a path from a given city to another city using various search methods we have discussed in class. The search methods you will implement for this project are breadth-first search, depth-first search, and A* search. This project will be implemented in Java and the main program will be in a file called `Search.java`.

Project Requirements

Look for the file `Proj01.zip` posted to myCourses. Download and unzip this file.

A list of cities from `Map.pdf` is contained in the file `city.dat`. Each row in this file corresponds to one city on the map, in the format: CITY STATE LATITUDE LONGITUDE. Each city has a direct route to some of the other cities on the map (you may assume that a "direct route" means a straight path to the city). Thus, the map can be represented by a graph, where the vertices of the graph are cities and the edges of the graph are the direct routes between cities. A list of the direct routes between any two cities is given in the file `edge.dat`. Each row in this file corresponds to a direct route connecting two cities, in the format CITY1 CITY2. Note that if there is a direct route from city1 to city2, then there should also be a direct route from city2 to city1. The edge file only gives the route from city1 to city2, but it should be easy to get the route from city2 to city1.

Your main program will be in a file called **`Search.java`**. The names of the .dat files (`city.dat` and `edge.dat`) should be hard-coded into your program (not read from the command line). However, your program will accept two command line arguments - an input file and an output file. The input file will consist of two lines: the first line is the name of the start city and the second line is the name of the destination city. The output file will consist of the results from running your program: the path from the start city to the destination city for each of the three search algorithms (breadth-first, depth-first, and A*). A sample input and output file are given in the next section.

After reading `city.dat` and `edge.dat`, your program should run each of the three search algorithms. After each run, print the list of cities on the path, the total number of "hops" it took to reach the destination city for that algorithm (a hop is just one edge on the path), and the total distance along the path.

When finding the paths for breadth-first and depth-first search, if there is a choice of cities to go to from any particular city, choose cities in alphabetical order. For BFS, sort neighbors in ascending order by name. For DFS, sort neighbors in descending order by name. (See sample output below for example). For A* search, if there is a choice of cities, choose cities according to the heuristic of how far away each city is from the destination city ("as the crow flies" - i.e., Euclidean distance) plus the distance from the start city to the current city under consideration. The LATITUDE and LONGITUDE fields in the `city.dat` file will be helpful for finding the distance. Note that latitude and longitude can be thought of as Cartesian (x-y) coordinates, where the origin is in the lower right-hand corner, with increasing latitude upward and increasing longitude leftward.

To convert from latitude and longitude to distance in miles, use the following formula:

```
distance = sqrt( (lat1-lat2)*(lat1-lat2) + (lon1-lon2)*(lon1-lon2) ) * 100
```

Sample Output

Below is an example of what your output should look like, assuming the start city is Olympia and the destination city is SaltLakeCity. Notice that if the destination city is on the fringe, search will stop before any other cities on the fringe are expanded. Your output should look identical to this output, including spacing.

```
Breadth-First Search Results:
Olympia
Boise
SaltLakeCity
That took 2 hops to find.
Total distance = 1264 miles.


Depth-First Search Results:
Olympia
Boise
SaltLakeCity
That took 2 hops to find.
Total distance = 1264 miles.


A* Search Results:
Olympia
Boise
SaltLakeCity
That took 2 hops to find.
Total distance = 1264 miles.
```

You can check your output for correct formatting by saving this sample input file called "in1", generating your own output file by running your program with this input file, and then running the UNIX `diff` command to compare your output with the expected output file called "out1":

```
$ java Search in1 output
$ diff -w output out1
```

If nothing prints to the screen after this command then your output is formatted correctly. You may also test your code on "in2" and "out2", for finding a path between Denver and Boston.

Your program will be tested on multiple start and destination cities, not just the samples shown here, so be sure to fully test your program, including all error conditions (given below). Also, your program should be able to accept input from standard input and/or print output to standard output instead of from files. A '-' in the command line will signify standard input and/or standard output. For example, your program should be able to be run as any of the following:

```
$ java Search - output
$ java Search input -
$ java Search - -
$ java Search input output
```

Error Conditions and Messages

1. Missing one or both command line arguments:
   - `Usage: java Search inputFile outputFile`
2. Cannot read `city.dat`:
   - `File not found: city.dat`
3. Cannot read `edge.dat`:
   - `File not found: edge.dat`
4. Cannot read input file:
   - `File not found: (input file name)`
5. Start city and/or destination city not in data file:
   - `No such city: (city)`

For the last two error messages, the string in parentheses should be replaced with the actual string that the user entered but was not found. Your program should print the error messages to `System.err` and terminate (`System.exit(0)`) upon any of these error conditions. No other error conditions will be tested for.

Hints

For DFS, if you use recursion, your order or nodes visited may not match the sample output, so I would recommend starting with an iterative approach.

For DFS, remember that the path that you obtain may not be optimal.

For DFS remember to sort nodes in reverse order so that when you pop off the stack, the order shown is preserved.

Other Requirements

Your submitted code must run on the CS machines (such as queeg or glados), so be sure to test your code in the target environment prior to submission.

Your program should use files found in the current directory ("./").

Submission

Take all of your `.java` source code files along with an optional `README.txt` file and create a `.zip` file. Submit ONLY this `.zip` file to the MyCourses dropbox for **Project 01** before 11:59pm on the date given at the top of this write-up. The same dropbox will remain open until 24 hours after the due date for late submissions (20% penalty), after that **no late submissions will be accepted for any reason .**