CS 452/552: Assignment 1

Due: October 9, 2023, by 11:00 PM (Central)

Overview

Your goal for this assignment is to write a program that reads in an input file describing a set of cities and then searches for paths between two user-specified cities using various search strategies. Program requirements can be found starting on page 2.

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "A* in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must not make use of any non-standard or third-party libraries.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**

Program Requirements

Your program should be runnable from the command line, and it should be able to process command-line arguments to update various program options and parameters as needed. The command-line arguments are listed below:

- `-f <FILENAME>`: Reads city data from the text file named `<FILENAME>` (specified as a `String`); note that the filename may include path information, so you should not assume that the file is located in the same directory as your source code. File format details are on page 3.
- `-i <STRING>`: Specifies the initial city as a `String`; multi-word city names must be enclosed with quotation marks (e.g., `-i "La Crosse"`).
- `-g <STRING>`: Specifies the goal city as a `String`; multi-word city names must be enclosed with quotation marks (e.g., `-g "La Crosse"`).
- `-s <STRING>`: Specifies the search strategy to be used; `<STRING>` should be one of:
  - `a-star`: Use the **A* search** strategy
  - `greedy`: Use the **greedy best-first search** strategy
  - `uniform`: Use the **uniform-cost search** strategy

  The next two strategies are only required for CS 552 students:
  - `breadth`: Emulate **breadth-first search** (specifically, **graph search** with a queue)
  - `depth`: Emulate **depth-first search** (specifically, **graph search** with a stack)

  If the `-s` argument is not provided, then A* search should be used by default.
- `-h <STRING>`: Specifies the heuristic function to be used (if applicable); `<STRING>` should be one of:
  - `haversine`: Use the **Haversine formula**
  - `euclidean`: Use the **Euclidean distance**

  If the `-h` argument is not provided, then the Haversine formula should be used by default; see the **Node Evaluation** section for further details.
- `--no-reached`: Disables the use of a reached table in the search algorithm, resulting in a tree-like search with redundant paths (including cycles). If this argument is not provided, then the search algorithm should use a reached table for removing redundant paths.
- `-v <INTEGER>`: Specifies a verbosity level indicating how much output the program should produce; default is `0` (see the **Output** section for details).

The `-f`, `-i`, and `-g` options (with arguments) are required; the remaining ones are optional (meaning that they might not be specified when the program is run, **not** that you don't have to support them!). You can assume that your program will only be run with valid arguments (so you do not need to include error checking, though it may be helpful for your own testing). Your program must be able to handle command-line arguments in **any** order; e.g., do not assume that the first argument will be `-f`, do not assume that the `-i` flag will appear before the `-g` flag, etc.

For example, if you write a Java program with your `main` method in a class named `Search`, then your program could be run as:

```
shell$ java Search -f cities01.csv -i "La Crosse" -g Minneapolis
```

A more detailed run might specify additional options:

```
shell$ java Search -g Winona -f cities01.csv -s greedy -i "La Crosse" --no-reached
```

Any program that does not work this way will lose points from the assignment grade. In particular, your program should not require that the source code be edited (and possibly re-compiled) in order

to run with different inputs, and your program should not ask the user for any information via standard input after it starts running.

After processing command-line arguments, your program should open the specified file and load the associated data for city names, locations, and distances. If the initial and goal cities are not found in the input file, then the program should print a message indicating this and terminate.

File Format

A city navigation file will have the following format:

```
# Cities: name, latitude, longitude
<CITY NAME 1>, <LAT 1>, <LON 1>
  ...
<CITY NAME N>, <LAT N>, <LON N>
# Distances: name1, name2, distance
<CITY NAME U1>, <CITY NAME V1>, <MILES BETWEEN CITY U1 AND CITY V1>
  ...
<CITY NAME UM>, <CITY NAME VM>, <MILES BETWEEN CITY UM AND CITY VM>
```

Lines starting with `#` are comments, which may contain a section header (e.g., `# Cities`) or be irrelevant to program functionality (in which case they should be ignored by your program). The first segment of non-comment lines in the file describes a set of N cities, with fixed (latitude, longitude) coordinates and each city on a separate line. City names may contain multiple words or expressions, but not commas (which are used to separate the entries on a line).

The next segment of non-comment lines (after the `# Distances` section header) describes distances between pairs of named cities. 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 `cities01.csv` included in the `a01-data.zip` archive on Canvas:

```
# Cities: 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: name1, name2, distance
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
```

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 no distance given between La Crosse and Minneapolis, meaning that any path between those cities must pass through one of the others first.

Search Process

Once the cities are loaded, your program should search for a path from the initial city to the goal city by building a search tree starting from the initial city. Your program should follow the general outline of the **Best First Search** (BFS) strategy presented at the beginning of Lecture 02-5. Nodes on the frontier should be expanded in the order dictated by the node evaluation function f used in the specified search strategy (which is A* by default); see the **Node Evaluation** section for additional details. (**Hint for CS 552 students:** You can **emulate** breadth-first search and depth-first search within a general BFS strategy by using an appropriate node evaluation function f; you should **not** need to create a completely separate search process!)

Redundant paths should be removed through the use of a reached table, unless this is disabled through the `--no-reached` command-line argument. Your program should **not** do an explicit cycle check, though: using a reached table will automatically eliminate cycles, while omitting the reached table should allow cycles to be explored.

When the search process is complete, your program should print out the final path that it finds from the initial city to the goal, or `NO PATH` if no such path exists. Following that, your program should print:

- The total distance along that path (or `-1` if no path exists);
- The number of nodes that were generated in the search tree (i.e., the number of nodes ever placed into the frontier; nodes that are "pruned" through the use of the reached table should not be counted);
- The number of nodes remaining on the frontier at the end of search (these are the nodes that were never expanded).

More details regarding program output along with examples can be found starting on page 6.

For full credit, your code should be efficient enough that it can handle both the small and large sample files (`cities01.csv` and `cities02.csv`, respectively) in a reasonable amount of time.

Node Evaluation

For any node n in the search tree, the cost function $g(n)$ is the sum of the action costs from the root to node n. Additionally, the heuristic function $h(n)$ is given by either the **Haversine formula** or **Euclidean distance**, as specified with the `-h` flag.

If the **Haversine formula** is used, then $h(n)$ should be computed as the length of the shortest possible arc between the city associated with node n and the goal city on the surface of the Earth. This is guaranteed to be an admissible and consistent heuristic for the `cities01.csv`, `cities02.csv`, and `romania.csv` files given, if calculated correctly. This arc length d can be calculated using the **Haversine formula**, which uses the latitudes and longitudes of the two locations to compute d through the sequence of calculations shown below:

$$\begin{aligned}
\Delta_{\text{lon}} &= \text{lon}_2 - \text{lon}_1 \\
\Delta_{\text{lat}} &= \text{lat}_2 - \text{lat}_1 \\
a &= \left(\sin\left(\frac{\Delta_{\text{lat}}}{2}\right)\right)^2 + \cos(\text{lat}_1) \cdot \cos(\text{lat}_2) \cdot \left(\sin\left(\frac{\Delta_{\text{lon}}}{2}\right)\right)^2 \\
c &= 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \\
d &= R \cdot c \text{ (where } R \text{ is the radius of the Earth)}
\end{aligned}$$

Note: for these calculations, the inputs to the trigonometry functions (sin, cos, and atan2) must be in the form of **radians**, **not degrees**. Because the data given in the input file has latitude and longitude in degrees, you will need to convert to radians. Also, because distances in the input are given in miles, you should also use miles for the radius of the Earth (which is $3,958.8$ miles).

If the **Euclidean distance** is used, then $h(n)$ should be computed by interpreting each city's latitude and longitude as (x, y) coordinates in the Cartesian plane and then calculating $h(n)$ as the standard Euclidean distance between the (x, y) coordinates of the city associated with node n and the (x, y) coordinates of the goal city.

Note that for the `cities01.csv`, `cities02.csv`, and `romania.csv` files, the Euclidean distance will be in the **wrong units** when compared to the road distances given, as one degree of latitude is not equal to one mile. Additionally, this heuristic is **not admissible** (e.g., cities at $(0, -179)$ and $(0, 179)$ are "nearby" on the surface of the earth but not nearby in the Cartesian plane). This is fine, as the Euclidean distance is intended primarily to be used for **testing on your own city data**. As a concrete example, the following simple example specifies (x, y) coordinates for four "cities" using the latitude and longitude fields, along with several edges connecting them.

```
# Cities: name, x, y
# Simple example
# C----D
# |   /
# A--B
A, 0, 0
B, 2, 0
C, 0, 2
D, 3, 2
# Distances: name1, name2, distance
A, B, 2.5
A, C, 2.9
B, D, 4.2
C, D, 3.4
```

If city D is the goal, then the $h(n)$ values will be computed as:

$$h(n) = \begin{cases} \sqrt{13} \approx 3.61 & \text{if node } n\text{'s state is } A \\ \sqrt{5} \approx 2.24 & \text{if node } n\text{'s state is } B \\ 3 & \text{if node } n\text{'s state is } C \\ 0 & \text{if node } n\text{'s state is } D \end{cases}.$$

With a small example like the above, it is relatively easy to work through the search process for different search strategies **by hand** and compare it with the behavior of your program. It is **strongly recommended** that you create additional test cases to check corner-case behavior and ensure that your program is functioning as expected!

Output

The level of output produced by the program is controlled by the `-v` flag (for *verbosity*). The levels range from 0 to 3, with 3 containing the most output; the output produced is additive, meaning that the output for level i should also be produced for any level j with $j \geq i$. The desired output for each level is discussed in more detail below.

Your own output should be very similar to the example output shown below when your program is run with the same arguments; while the number of nodes encountered during search may be somewhat different depending upon implementation, they should be within the same order of magnitude. The distances along the paths found (and in most cases the actual path itself) should be the same.

Additional output can be found on Canvas in the `a01-example-output.zip` archive which may be helpful in debugging and/or checking your own implementation of some of the search strategies.

Output Level 0

Level 0 output is the default and just prints the overall route found (or `NO PATH`) along with some basic search information (see page 4 for details). An example is shown below:

```
shell$ java Search -f cities01.csv -i "La Crosse" -g Minneapolis
Route found: La Crosse -> La Crescent -> Minneapolis
Distance: 147.0

Total nodes generated      : 4
Nodes remaining on frontier: 0
```

Output Level 1

Level 1 output adds basic information about the search problem being solved along with information on the solution node found (if any) and the time taken to complete the search; this information gets printed before the final output. For the solution node n, you should print the node's state (city), the node's parent's state (or null if the node has no parent), and the node's $f(n)$, $g(n)$, and $h(n)$ values. An example is shown below:

```
shell$ java Search -f cities01.csv -i "La Crosse" -g Minneapolis -v 1
* Reading data from [cities01.csv]
* Number of cities: 4
* Searching for path from La Crosse to Minneapolis using A-Star Search
* Goal found  : Minneapolis  (p-> La Crescent)  [f= 147.0; g= 147.0; h=   0.0]
* Search took 7ms

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

Total nodes generated      : 4
Nodes remaining on frontier: 0
```

(You can use `System.currentTimeMillis()` to get the current time as a `long` type in Java.)

Output Level 2

Level 2 output adds information about the nodes being expanded during the search process itself:

```
shell$ java Search -f cities01.csv -i "La Crosse" -g Minneapolis -v 2
* Reading data from [cities01.csv]
* Number of cities: 4
* Searching for path from La Crosse to Minneapolis using A-Star Search
  Expanding   : La Crosse    (p-> null)         [f= 129.2; g=   0.0; h= 129.2]
  Expanding   : La Crescent  (p-> La Crosse)    [f= 130.6; g=   5.0; h= 125.6]
  Expanding   : Winona       (p-> La Crosse)    [f= 132.9; g=  31.6; h= 101.3]
  Expanding   : Minneapolis  (p-> La Crescent)  [f= 147.0; g= 147.0; h=   0.0]
* Goal found  : Minneapolis  (p-> La Crescent)  [f= 147.0; g= 147.0; h=   0.0]
* Search took 8ms

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

Total nodes generated      : 4
Nodes remaining on frontier: 0
```

Output Level 3

Level 3 output adds information about the child nodes that get generated during node expansion, including whether they are added to the frontier or not:

```
shell$ java Search -f cities01.csv -i "La Crosse" -g Minneapolis -v 3
* Reading data from [cities01.csv]
* Number of cities: 4
* Searching for path from La Crosse to Minneapolis using A-Star Search
  Expanding   : La Crosse    (p-> null)         [f= 129.2; g=   0.0; h= 129.2]
    Adding    : Winona       (p-> La Crosse)    [f= 132.9; g=  31.6; h= 101.3]
    Adding    : La Crescent  (p-> La Crosse)    [f= 130.6; g=   5.0; h= 125.6]
  Expanding   : La Crescent  (p-> La Crosse)    [f= 130.6; g=   5.0; h= 125.6]
    Adding    : Minneapolis  (p-> La Crescent)  [f= 147.0; g= 147.0; h=   0.0]
    NOT adding: Winona       (p-> La Crescent)  [f= 133.8; g=  32.5; h= 101.3]
    NOT adding: La Crosse    (p-> La Crescent)  [f= 139.2; g=  10.0; h= 129.2]
  Expanding   : Winona       (p-> La Crosse)    [f= 132.9; g=  31.6; h= 101.3]
    NOT adding: Minneapolis  (p-> Winona)       [f= 147.6; g= 147.6; h=   0.0]
    NOT adding: La Crescent  (p-> Winona)       [f= 184.7; g=  59.1; h= 125.6]
    NOT adding: La Crosse    (p-> Winona)       [f= 192.4; g=  63.2; h= 129.2]
  Expanding   : Minneapolis  (p-> La Crescent)  [f= 147.0; g= 147.0; h=   0.0]
* Goal found  : Minneapolis  (p-> La Crescent)  [f= 147.0; g= 147.0; h=   0.0]
* Search took 12ms

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

Total nodes generated      : 4
Nodes remaining on frontier: 0
```