Homework 1 — Using A* Search

UWL CS452/552 Artificial Intelligence

Fall 2019

For your first homework, please write a Java[1] class which implements A* search to find the best travel path betwee two cities, given cost information between a subset of pairs of those cities.

In the resources for this homework linked from Canvas, you will find two Java source files. One of these classes, `FoundPath`, just bundles up the details of an A* search. The other class, `Pathfinder`, is an abstract class which you will extend. The core of your work is the method `getPathInfo`[2]:

```
public abstract FoundPath getPathInfo(String startCity, String endCity);
```

Your class `AStar` should extend `Pathfinder`, implementing an A* search to find the shortest path between the named start- and endpoints. Its result bundles up certain information which your search should gather:

- The sequence of cities in the best path between cities, or an empty list if no such path is possible. The first element of this list should be `startCity`, and the last element of this list should be `endCity`.
- The total cost of following the above path, or the `empty` instance if there no such path is possible[3].
- The number of nodes which A* generated in the search (i.e. nodes ever placed into the frontier).
- The number of nodes left in the frontier at the end of the search (i.e. nodes in the frontier but never expanded).

The `FoundPath` interface describes methods for these values, and your implementation of `get-PathInfo` should return an object implementing this interface.

Your class `Pathfinder` should provide (among any other constructors which you might use for your own development and testing purposes) a one-argument constructor, where the argument is a `String` naming a file containing information about the cities known to your algorithm. Such a file will have the format shown in Figure 1. 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

```
# 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>
```

Figure 1: The cities specification file.

```
# 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
```

Figure 2: A small example of a cities specification.

through some other city first (if any such path exists). Figure 2 shows a small example with four local cities. This small file will be included in the examples to be made available on Canvas.

For your A* heuristic, you should use the length of the shortest possible arc on the surface of the Earth between a city C and the goal G.[4] If calculated correctly, this arc length is guaranteed to be an admissible and consistent heuristic for the data sets given.

Your code should be efficient enough that it can handle the larger example file which you will receive, which contains data for several hundred cities. However, small examples are typically easier for debugging. You are of course encouraged to make additional examples for your own purposes.

You should assume that the `getPathInfo` will be called more than once per instance. It should return consistent results for the same information from call to call.

We will see in class that the most basic specification of A* allows a single city to appear multiple times during a search path. 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), generating these paths is inefficient, and we will want to avoid it. There will be two submission points for your code:

- Your first implementation should leave this inefficiency in, so repeated nodes are allowed during search.
- Your second submission should remove this inefficiency, so that repeated nodes are ignored

along a single path.

Your code should be well-formatted, and documented with both Javadoc and inline comments. The inline comments should focus on the explanation of the AI ideas implemented.

There are a number of other abstract methods defined in `Pathfinder.java`, and a number of additional requirements mentioned in the Javadoc comments of that file, which I expect you to implement.

The two implementations for this assignment are due on Thursday, September 26, at the usual deadline time of 11:59pm (La Crosse local time). Do *not* resubmit the `FoundPath.java` and `Pathfinder.java` files, which you must not alter. Details of the exact form and mechanics of the submission are to follow.

Notes

1. I prefer that you code the solution in Java, but I am open to discussing other languages. Come speak with me before you start, so we can discuss feasibility and agree on a protocol.

   Two points related to choosing an alternative language:

   - Do not wait to discuss this: because we may need time to think about the format of your work, and because I can manage only so many other languages.
   - I have not yet written all of the homeworks for this class, and I cannot guarantee that every homework will support alternative languages.

2. The full file also contains Javadoc comments, additional declarations, and helper/administrative methods — see the full source for details.

3. The `java.util.Optional` class represents a value which *could* be present, or which might not exist at all. See its Javadoc API documentation for more information, and for the static methods used to create an `Optional` instance.

4. Details of how to calculate this distance using the *Haversine formula* can be found at *http://andrew.hedges.name/experiments/haversine* . Note that for this formula 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.
   - Since distances in the input are given in miles, you should also use miles for the radius of the Earth.

   Do not worry about possible errors in the heuristic arising from the fact that our Earth is not a perfect sphere.