COGS 125 / CSE 175
Introduction to Artificial Intelligence
Specification for Programming Assignment #1

David C. Noelle

Due: 11:59 P.M. on Monday, October 16, 2017

Overview

This programming assignment has three main learning goals. First, the assignment will provide you with an opportunity to practice your skills in developing programs in Java™ using the Eclipse integrated development environment (IDE). Second, this assignment will provide you with some experience in implementing basic heuristic search algorithms, including *greedy best-first search* and *A* search*. For comparison, you will also implement the uninformed *uniform-cost search* algorithm. Third, this assignment requires you to design an admissible heuristic function in the domain of searching for a shortest path on a map. This will provide you with some experience exploring the features that make for good heuristic functions. A solid understanding of these basic approaches to heuristic search will form an important foundation for knowledge of the more advanced techniques to be covered throughout this class.

In summary, you will implement *uniform-cost search*, *greedy best-first search*, and *A* search* algorithms in Java™ in the context of a simple shortest-path map search problem. These implementations will also support the optional checking for repeated states during search. You will also design an admissible heuristic function, with the goal of demonstrating a substantial improvement in performance over uninformed search methods.

Submission for Evaluation

To complete this assignment, you must generate and provide four Java™ class source files: "`GoodHeuristic.java`", "`UniformCostSearch.java`", "`GreedySearch.java`", and "`AStarSearch.java`". The first of these files should implement an extention of the provided `Heuristic` class, called `GoodHeuristic`, which implements a good admissible heuristic function for the map search problem. A skeletal template for this file will be provided to you. The remaining three files should implement the search algorithms referenced in their names, as described below. These are the only four files that you should submit for evaluation.

To submit your completed assignment for evaluation, log onto the class site on CatCourses and navigate to the "Assignments" section. Then, locate "Programming Assignment #1" and select the option to submit your solution for this assignment. Provide your four program files as four separate attachments. Do not upload any other files as part of this submission. Comments to the teaching team should appear as header comments in your Java™ source code files.

Submissions must arrive by 11:59 P.M. on Monday, October 16th. Please allow time for potential system slowness immediately prior to this deadline. You may submit assignment solutions multiple times, and only the most recently submitted version will be evaluated. As discussed in the course syllabus, late assignments will not be evaluated.

If your last submission for this assignment arrives by 11:59 P.M. on Friday, October 13th, you will receive a 10% bonus to the score that you receive for this assignment. This bonus is intended to encourage you to try to complete this assignment early.

Activities

You are to provide Java™ classes that implement the following three search algorithms: uniform-cost search, greedy best-first search, and A* search. Your provided Java™ source code must be compatible with a collection of provided Java™ classes that implement simple road maps, allowing your search algorithms to be used to find the shortest routes between locations on such maps. Indeed, your assignment solution will be evaluated by combining your submitted class files with copies of the provided map utility class files and testing the resulting complete program against a variety of test cases. In other words, *your solution must work with the provided utilities, without any modifications to these provided files*.

More specifically, you are to provide the following classes in the corresponding files, implementing the corresponding algorithms:

Class	**File**	**Algorithm**
`UniformCostSearch`	"`UniformCostSearch.java`"	uniform-cost search
`GreedySearch`	"`GreedySearch.java`"	greedy best-first search
`AStarSearch`	"`AStarSearch.java`"	A* search

The source code for each of these classes should be very similar to that for the others. In fact, all of these classes must have the following features ...

- a constructor that takes four arguments:
  1. a complete Map object, encoding the map to be searched
  2. a String providing the name of the starting location
  3. a String providing the name of the destination location
  4. an integer depth limit for the search — if this depth is ever reached during a search (i.e., a node at this depth or deeper is considered for expansion), the search should be abandoned and `null` (indicating search failure) should be returned

- a method that actually performs the search, called `search`, with the following properties:
  - it takes a single boolean argument — if this argument is "true", then repeated state checking should be performed, otherwise no such checking should be done during search
  - it returns a Waypoint object from the search tree that corresponds to the target destination, or `null` if no solution was found
- an integer instance variable called `expansionCount` that contains the number of node expansions performed during the last call to the `search` method

In general, your classes should allow the `main` method in the provided `Pone` class to output correct solutions (including path cost and expansion count statistics) for any map search problem provided as input to it.

In addition to these search algorithm classes, you must also provide a utility class, called `GoodHeuristic`, in a file named "`GoodHeuristic.java`" that extends the `Heuristic` class (see below) and implements an admissible heuristic function that can be quickly calculated. You are required to design this heuristic function by yourself, and the quality of your heuristic function will strongly influence how your submitted assignment is evaluated. Your heuristic function must absolutely be admissible, but it should otherwise reflect as accurate an estimate of the residual path cost from a given search tree node as possible, given the constraint of rapid calculation. Note that `Location` objects contain `longitude` and `latitude` fields that may assist in this process. Also note that, for the purpose of this assignment, the cost assigned to road segments is to be taken as a *time cost*. In other words, the goal of the search is to find the shortest path in terms of travel time, and each road segment is labeled with the time it takes to traverse that segment. This means that any measure of physical distance will *not* suffice as an admissible heuristic function, as such measures do not reflect an estimate of the remaining *travel time* to the destination. If necessary, you may assume that location `longitude` and `latitude` values are Cartesian coordinates measured in miles, and road segment costs are expressed in minutes. If you make such an assumption, however, you should indicate this fact in a comment in your submitted "`GoodHeuristic.java`" file.

The Java™ utility classes that you are required to use are provided in a ZIP archive file called "`PA1.zip`" which is available in the "Assignments" section of the class CatCourses site, under "Programming Assignment #1". These utilities include:

- `Location` — This object encodes a location on the map. It is a "state" in the "state space" to be searched.
- `Road` — This object encodes a road segment from one location to another. Each `Location` records all of the road segments leading away from that location. This list of `Road` objects provides the "successor function" result for each "state".
- `Map` — This object gathers together all of the `Location` objects on the map. It provides utilities for reading map descriptions from files.

- `Waypoint` — This object encodes a "node" in the search tree. Each `Waypoint` represents a `Location` in the context of a particular path from the starting point. Note that these objects also record the partial path cost and the heuristic function value associated with the given search tree node.

- `Frontier` — This object provides a simple implementation of queues and stacks containing `Waypoint` objects. This can be used to keep track of the nodes in a search tree's "frontier" or "fringe". Note that this class is provided only for your reference, and *it should not be used for this assignment*. Instead, some form of priority queue is needed to implement the search tree's "frontier" in the search algorithms to be implemented in this assignment, and the next class, described below, implements such a priority queue.

- `SortedFrontier` – This object provides a simple implementation of priority queues containing `Waypoint` objects. When search tree nodes are to be removed from the "frontier" in an order specified by some real-valued numeric ranking, as is needed for all three of the search algorithms to be implemented here, this class provides an ideal way to keep track of the nodes in a search tree's "fringe". Objects of this type maintain a "sorting strategy" field which indicates the statistic of the contained `Waypoint` objects that should be used for sorting nodes. The nodes can be sorted by partial path cost, by heuristic function value, or by f-cost.

- `Heuristic` – This object provides a way for heuristic functions to be recorded and passed as arguments to other functions. Heuristic function objects keep track of the current search's destination location, and they provide a method that assigns a real-valued heuristic value to any given `Waypoint` object, given that target destination. The basic `Heuristic` class assigns a zero value to all search tree nodes, which is an admissible heuristic function, but not a very useful one.

- `GoodHeuristic` – This class extends the `Heuristic` class, allowing you to override the useless heuristic provided by that parent class with something that is a better estimate of the residual path cost from a given node. You are free to cache any information you like in `GoodHeuristic` objects, and you should strive to make your heuristic function as accurate as possible while still remaining admissible and quick to calculate. You are required to submit a completed copy of this provided file for evaluation.

- `Pone` — This object provides a top-level driver that tests your search algorithms, with repeated state checking both turned on and turned off. Your code must allow `Pone` to produce correct output.

The contents of these Java™ utility files will be discussed during a course laboratory session, and inline comments in these files should assist in your understanding of the provided code. Questions are welcome, however, and should be directed to the teaching team.

Your implementations of all three search algorithms should make use of the general search algorithm presented as "`GRAPH-SEARCH`" in the course textbook, Russell & Norvig (2010). Specifically, your code must test for goal attainment just prior to expanding a node (*not* just prior to

insertion into the frontier), and repeated state checking must be performed as nodes are inserted into the frontier (*not* upon removal from the frontier). When repeated state checking is being done, a child node should only be discarded if its state matches that of a previously expanded node or of a node currently in the frontier. (To be clear, a child node with a state that matches that of a node currently in the frontier may or may not be discarded due to repeated state checking, depending on the specific algorithm being implmented and the relative "costs" of the two nodes.) Note that the "`GRAPH-SEARCH`" algorithm will require a slight modification to allow for the *disabling* of repeated state checking, based on the boolean argument provided to the `search` function. In general, your implementations should *not* depend on recursion to traverse the search tree, and they should make explicit use of a `SortedFrontier` object to keep track of the fringe.

Your submission will be evaluated primarily for accuracy, with efficiency being a secondary consideration. Your source code *will* be examined, however, and the readability and style of your implementation will have a substantial influence on how your assignment is evaluated. As a rough rule of thumb, consider the use of good software writing practices as accounting for approximately 10% to 20% of the value of this exercise. Note also that, as discussed in the course syllabus, submissions that fail to successfully compile under the laboratory Eclipse Java™ IDE (i.e., they produce "build errors") will not be evaluated and will receive *no credit*.

As for all assignments in this class, submitted solutions should reflect the understanding and effort of the individual student making the submission. Not a single line of computer code should be shared between course participants. If there is ever any doubt concerning the propriety of a given interaction, it is the student's responsibility to approach the instructor and clarify the situation *prior* to the submission of work results. Also, helpful conversations with fellow students, or any other person (including members of the teaching team), should be explicitly mentioned in submitted assignments (e.g., in comments in the submitted source code files). Failure to appropriately cite sources is called *plagiarism*, and it will not be tolerated!

The members of the teaching team stand ready to help you with the learning process embodied by this assignment. Please do not hesitate to request their assistance.