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

David C. Noelle

Due: 11:59 P.M. on Friday, October 20, 2023

Overview

This programming assignment has three main learning goals. First, the assignment will provide you with an opportunity to practice your skills developing Python programs in the PyCharm 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 foundational understanding of these basic approaches to heuristic search will support your learning of more advanced techniques later in this class.

In summary, you will implement the *uniform-cost search*, *greedy best-first search*, and *A* search* algorithms in Python in the context of a simple shortest-path map search problem. These implementations will also support the optional checking of 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 four Python script files: "`heuristic.py`", "`ucost.py`", "`greedy.py`", and "`astar.py`". The first of these files should implement 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 Python source code files.

Submissions must arrive by 11:59 P.M. on Friday, October 20th. 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 and will receive *no credit*.

If your last submission for this assignment arrives by 11:59 P.M. on Tuesday, October 17th, 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 Python functions that implement the following three search algorithms: *uniform-cost search*, *greedy best-first search*, and *A* search*. Your provided Python source code *must* be compatible with provided Python utility code which implements 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 files with copies of the provided utility 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 functions in the corresponding files, implementing the corresponding algorithms:

Function	**File**	**Algorithm**
`uniform_cost_search`	"`ucost.py`"	*uniform-cost search*
`greedy_search`	"`greedy.py`"	*greedy best-first search*
`a_star_search`	"`astar.py`"	*A* search*

The source code for each of these functions should be very similar to that for the others. Rough pseudocode for these functions has been provided during class meetings. These functions must have the following features . . .

- takes two or three arguments:
  1. `problem` — a `RouteProblem` object
  2. `h` — a `HeuristicFunction` object (not included in `uniform_cost_search`)
  3. `repeat_check` — a boolean indicating if repeated state checking is to be done
- implements the pseudocode for generic search provided during class lectures
- deviates from this pseudocode only by performing repeated state checking if and only if the `repeat_check` argument is `True`

- in particular, the goal test is performed on a search tree node just before it is expanded
- makes use of a `Frontier` object to maintain the search tree fringe
- returns a search tree node corresponding to a solution, or `None` if no solution is found

In general, your functions should allow the provided "`main.py`" script to output correct solutions (including path cost and expansion count statistics) for *any* map search problem provided as input to it. Note that this means that the functions that you implement should write *no output*, as this will clutter the output produced by the "`main.py`" script. If you include any statements that write output in your code (perhaps as tools for debugging) these should be removed prior to submitting your code files for evaluation. You may receive *no credit* for your submitted solution if it produces extraneous output.

In addition to these search algorithm functions, you must also provide a definition for a class called `HeuristicFunction`, in a file named "`heuristic.py`", that 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 have a substantial influence on 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 locations have longitude and latitude coordinates 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 an *amount of gasoline needed* to traverse each road segment. In other words, the goal of the search is to find the shortest path in terms of the *total amount of gasoline used*, and each road segment is labeled with the amount of gas it takes to traverse that segment. (For example, a road segment cost of 4.2 might be interpreted as a need to burn 4.2 deciliters of gasoline to travel that road 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 *gasoline needed* to get to the destination. If you cannot think of a solution that is admissible regardless of the units of the map measurements, you may assume that location coordinates are measured in miles from an origin point and road segment costs are expressed in deciliters. If you make an assumption about the units of measurement, or any other assumptions, you should indicate this fact in a comment in your submitted "`heuristic.py`" file. (Better solutions to this assignment will not require such assumptions.) Keep in mind that your search algorithms should work for *any* valid map search problems provided as input to them.

The Python utility code that you are required to use is 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:

- `RoadMap` class — This class encodes a simple road map, involving locations connected by road segments. Each location has a name and coordinates, which can be conceived as longitude and latitude. Each road segment represents a one-way connection from one location to another. Each road segment has a name and a cost of traversal.

- `RouteProblem` class — This class encapsulates a formal specification of a search problem. It includes a `RoadMap`, as well as a starting location and goal location. It also provides a goal test function called `is_goal`.

- `Node` class — This class implements a node in a search tree. Each node has a corresponding location, a parent node, and a road segment used to get from the location of the parent to the location of the node. (Note that the root of the search tree, corresponding to the starting location, has no parent or road segment.) Each node also tracks its own depth in the search tree, as well as the partial path cost from the root of the tree to the node. Each node also records the value of the heuristic evaluation function applied to the node's location. Optionally, each node can record the heuristic function to use (see below), and children will inherit the function from their parents. The class provides an `expand` function, which expands the node.

- `Frontier` class — This class implements the frontier, or fringe, of a search tree. The root node of a search tree is provided upon creation, initially populating the frontier with that one node. These objects are implemented as priority queues, releasing nodes in order of increasing values of some measure. At the time of the creation, the measure to be used to sort the nodes in the frontier must be specified: 'g' (partial path cost), 'h' (heuristic evaluation function value), or 'f' (the sum of the other two measures). This class provides functions to `add` a node to the frontier and `pop` a node from the frontier, as well as testing if the frontier `is_empty` or if it `contains` a node with a matching location.

The contents of these utility files will be discussed during a laboratory session, and 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 largely mirror the generic search algorithm presented during class lectures. Specifically, your code must test for goal attainment just prior to expanding a node (*not* just prior to insertion into the frontier). No repeated state checking should be done unless the `repeat_check` argument is `True`. When repeated state checking is being done, a child node should only be discarded if its state matches that of a previously seen node. (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 algorithm presented in class will require a slight modification to allow for the *disabling* of repeated state checking, based on the boolean argument provided to the search function. Your implementations should *not* depend on recursion to traverse the search tree, and they should make explicit use of a `Frontier` object to keep track of the fringe.

In order to obtain some confidence that your search algorithms work for *any* valid map search problems provided as input to them, as required, it is very likely that you will have to test your solution on a variety of test cases. A simple test case appears in the "`main.py`" script, but this test case is insufficient to fully test your code. **It is possible for your code to contain serious errors but still perform well when using the provided example map.** Thus, part of this assignment includes coming up with as many distinct difficult search problems as possible and ensuring that

your solution produces appropriate output for all of them. Your test cases will not be submitted for evaluation, but the quality of your submitted Python source code files will likely depend on the breadth of testing that you perform.

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. Please use the coding practices exemplified in the provided utility files as a guide to appropriate readability and style. Note also that, as discussed in the course syllabus, submissions that fail to run without crashing on the laboratory PyCharm IDE 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. This is *not* a group assignment.** 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). These comments should also explicitly mention any written resources, including online resources, that were used to complete the exercise. Citations should clearly identify the source of any help received (e.g., "Dr. David Noelle" instead of "a member of the teaching team", "The Python Tutorial at `docs.python.org/3/tutorial/`" instead of "Python documentation"). **Failure to appropriately cite sources is called *plagiarism*, and it will not be tolerated!** Policy specifies that detected acts of academic dishonesty must result minimally with a zero score on the assignment, and it may result in a failing grade in the class. Please see the course syllabus for details.

To be clear, please note that all of the following conditions must hold for a submitted solution to receive *any* credit:

- solution submitted by the due date
- solution runs without crashing on the laboratory PyCharm IDE
- solution produces no extraneous output
- solution works with unmodified utility code, as provided
- solution does *not* include code written by another person (with or without modifications)
- solution explicitly cites all help received, whether from a person or some other source

While partial credit will be given for submissions that meet these conditions, failure to meet one or more of these conditions will result in *no credit* and, in the case of academic dishonesty, may result in a failing grade in the course. Please see the course syllabus for details.

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.

Uniform-Cost Search

```
function SEARCH(problem) returns a solution node or failure
  node <- a node containing the initial state of problem
  if node contains a goal state of problem then return node
  initialize the frontier to contain node
  initialize the reached set to contain node
  while frontier is not empty
    node <- a leaf node removed from frontier
    if node contains a goal state of problem then return node
    expand node
    for each child of node
      if child is in the reached set
        if child is in frontier but with a higher cost
          remove the matching node from frontier
          add child to frontier
      else
        add child to frontier
        add child to the reached set
  return failure
```

For the frontier, use a priority queue that sorts by $g(n)$ cost.

Greedy Search

```
function SEARCH(problem) returns a solution node or failure
  node <- a node containing the initial state of problem
  if node contains a goal state of problem then return node
  initialize the frontier to contain node
  initialize the reached set to contain node
  while frontier is not empty
    node <- a leaf node removed from frontier
    if node contains a goal state of problem then return node
    expand node
    for each child of node
      if child is in the reached set
        if child is in frontier but with a higher cost
          remove the matching node from frontier
          add child to frontier
      else
        add child to frontier
        add child to the reached set
  return failure
```

For the frontier, use a priority queue that sorts by *h(n)* cost.

A* Search

```
function SEARCH(problem) returns a solution node or failure
  node <- a node containing the initial state of problem
  if node contains a goal state of problem then return node
  initialize the frontier to contain node
  initialize the reached set to contain node
  while frontier is not empty
    node <- a leaf node removed from frontier
    if node contains a goal state of problem then return node
    expand node
    for each child of node
      if child is in the reached set
        if child is in frontier but with a higher cost
          remove the matching node from frontier
          add child to frontier
      else
        add child to frontier
        add child to the reached set
return failure
```

For the frontier, use a priority queue that sorts by f(*n*) cost.