CSCI-561 - Spring 2023 - Foundations of Artificial Intelligence
Homework 1

Due February 5, 2024 23:59

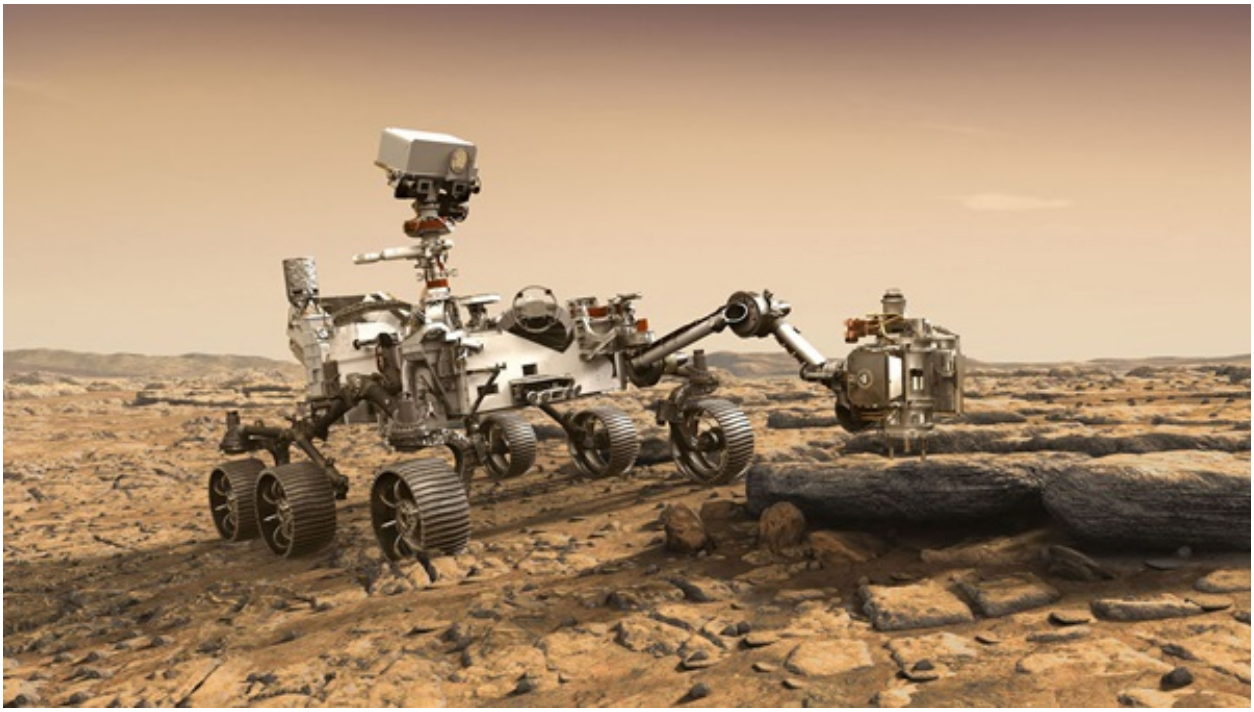

Image from https://www.astronomy.com/space-exploration/check-out-nasas-latest-mars-rover/

Guidelines

This is a programming assignment. You will be provided sample inputs and outputs (see below). The goal of the samples is to check that you can correctly parse the problem definitions and generate a correctly formatted output which is also a correct solution to the problem. For grading, your program will be tested on a *different* set of samples. It should not be assumed that if your program works on the samples it will work on all test cases, so you should focus on making sure that your algorithm is general and algorithmically correct. You are encouraged to try your own test cases to check how your program would behave in some corner cases that you might think of. Since **each homework is checked via an automated A.I. script**, your output should match the specified format ***exactly***. Failure to do so will most certainly cost some points. The output format is simple and examples are provided below. You should upload and test your code on vocareum.com, and you will also submit it there.

Grading

Your code will be tested as follows: Your program should not require any command-line argument. It should read a text file called “input.txt” in the current directory that contains a problem definition. It should write a file “output.txt” with your solution to the same current

directory. Format for input.txt and output.txt is specified below. End-of-line character is LF (since vocareum is a Unix system and follows the Unix convention).
The grading A.I. script will, 50 times:

- Create an input.txt file, delete any old output.txt file.
- Run your code.
- Check correctness of your program's output.txt file.
- If your outputs for all 50 test cases are correct, you get 100 points.
- If one or more test case fails, you get 100 – 2xN points where N is the number of failed test cases.

Note that if your code does not compile, or somehow fails to load and parse input.txt, or writes an incorrectly formatted output.txt, or no output.txt at all, or OuTpUt.TxT, or runs out of memory or out of time (details below), **you will get zero points**. Anything you write to stdout or stderr will be ignored and is ok to leave in the code you submit (but it will likely slow you down). Please test your program with the provided sample files to avoid any problem.

Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe you are urged to err on the side of caution. Do not copy work from another student or off the web. Keep in mind that sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

Do not copy code or written material from another student. Even single lines of code should not be copied.
Do not collaborate on this assignment. The assignment is to be solved individually.
Do not copy code off the web. This is easier to detect than you may think.
Do not share any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones considered below.
Do not copy code from past students. We keep copies of past work to check for this. Even though this problem differs from those of previous years, do not try to copy from homeworks of previous years.
Do not ask on piazza how to implement some function for this homework, or how to calculate something needed for this homework.
Do not post code on piazza asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.
Do not post test cases on piazza asking for what the correct solution should be.

Do ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

Project description

In this project, we look at the problem of path planning in a different way just to give you the opportunity to deepen your understanding of search algorithms and modify search techniques to fit the criteria of a realistic problem. To give you a context for how search algorithms can be utilized, we invite you Mars. Using a rover, we want to collect soil samples. Because the rover has a limited life expectancy, we want to travel from our current location to the next soil sample location as efficiently as possible, i.e., taking the shortest path. Using advanced satellite imaging techniques, the terrain of Mars has already been scanned, discretized, and simplified into a set of locations that are deemed safe for the rover, as well as a set of path segments that are safe between locations. This is similar to a standard route planning problem using a topological map of cities and street segments, except that here we also account for elevation (locations have 3D coordinates). One additional complication is that the rover has a limited amount of available motor energy on every move, such that it cannot travel on path segments that are going uphill above a certain limit. This, however, can be overcome in some cases if the rover has gained some momentum by going downhill just before it goes uphill. This is explained in more details below. For now, just beware that whether the rover can or cannot traverse an uphill path segment depends on whether it has been going downhill and has gained momentum just before.

Search for the optimal paths

Our task is to lead the rover from a start location to a new location for the next soil sample. We want to find a shortest path among the safe paths, i.e., one that minimizes the total distance traveled. There may be one, several, or no solutions on a given instance of the problem. When several optimal (shortest) solutions exist, your algorithm can return any of those.

Problem definition details

You will write a program that will take an input file that describes the terrain (lists of safe locations and of safe path segments), the starting location, the goal location, and the available motor energy for going uphill. Your program should then output a shortest path as a list of locations traversed, or FAIL if no path could be found. A path is composed of a sequence of elementary moves. Each elementary move consists of moving the rover from its current safe location to a new safe location that is directly connected to the current location by a single safe path segment. To find the solution you will use the following algorithms:

- Breadth-first search (BFS) with step cost of 1 per elementary move.
- Uniform-cost search (UCS) with 2D step costs (ignoring elevation).
- A* search (A*) with 3D step costs.

In all cases, you should consider that a path segment is only traversable if either it is not going uphill by more than the uphill energy limit, or the rover currently has enough momentum from the immediately preceding move to overcome that limit.

Terrain map

The terrain will be described as a graph, specified in input.txt using two lists:

- List of safe locations, each with a name and a 3D coordinate (x, y, z)
- List of safe path segments, each given as a pair of two safe location names.

Path segments are not directed, i.e., if segment "aaa bbb" is in the list, then the rover could in principle either travel from aaa to bbb or from bbb to aaa. Beware, however, that the uphill energy limit and momentum may prevent some of those travels (details below).

To avoid potential issues with rounding errors, your path will be considered a correct solution if its length is within 0.1 of the optimal path length calculated by our reference algorithm (and if it also does not violate any energy/momentum rules). We recommend using double (64-bit) rather than float (32-bit) for your internal calculations.

Energy and momentum

The required energy for a given move is always calculated from a point (x1, y1, z1) to another point (x2, y2, z2) as z2 – z1.

Likewise, momentum is here simply defined as the opposite of the energy of the previous move. For simplicity, we assume that momentum does not accumulate over several successive moves. Hence, momentum at a given point is only given by the downhill energy obtained from the last move that brought us to that point.

If the combined momentum from the previous move and required energy for the next move is exactly the energy limit, assume that the robot will be able to make that next move. To avoid issues with rounding errors, energy limit and location coordinates are integers, and hence momentum is too.

Algorithm variants

To help us distinguish between your three algorithm implementations, you must follow the following conventions for computing operational path length:

- **Breadth-first search (BFS)**

In BFS, each move from one location to a direct neighbor counts for a unit path cost of 1. You do not need to worry about the elevation levels or about the actual distance traveled. However, you still need to make sure the move is allowed according to energy and momentum. Therefore, any *allowed move* from one location to a graph-adjacent location costs 1. The length of any path will hence always be an integer.

- **Uniform-cost search (UCS)**

When running UCS, you should compute unit path costs in 2D, as the Euclidean distance between the (x1,y1) and (x2,y2) coordinates of two locations. You still need to make sure the move is allowed, in the same way you did for BFS. Path length is now a floating point number.

- **A* search (A*).**

When running A*, you should compute unit path costs in 3D, as the Euclidean distance between the (x1,y1,z1) and (x2,y2,z2) coordinates of two locations. You still need to make sure the move is allowed, in the same way you did for BFS and UCS. Path length is again a floating point number.

Remember: In addition to computing the path cost, you also need to design an admissible heuristic for A* for this problem.

File formats

Input: The file input.txt in the current directory of your program will be formatted as follows:

First line:	Instruction of which algorithm to use, as a string: BFS, UCS or A*
Second line:	One positive 32-bit integer specifying the rover's uphill energy limit.
Third line:	One strictly positive 32-bit integer for the number N of safe locations.
Next N lines:	A string on each line with format "name x y z" where name is a lowercase alphabetical string denoting the name of the location x, y, and z are 32-bit signed integers for the location's coordinates You are guaranteed that exactly one of the N names will be "start", and exactly one will be "goal". These are the starting and goal locations.
Next line:	One strictly positive 32-bit integer for the number M of safe path segments.
Next M lines:	A string on each line for safe path segments with format "nameone nametwo" where nameone and nametwo are the names of two locations, each guaranteed to exist in the previously provided set of N locations.

For example:

```
BFS
23
4
start 0 0 0
abcd 1 2 3
efgh 4 5 6
goal 8 9 10
3
start abcd
abcd efgh
efgh goal
```

You are guaranteed that the format of input.txt will be correct (e.g., if the file specifies that N is 4, you can assume that the following 4 lines always exist and are correct definitions of safe locations). There will be no duplicate safe path segments (e.g., if "abcd efgh" is specified in the list of paths, it will only appear once, and the equivalent "efgh abcd" will not appear in the list). Finally, two safe locations will never have the same coordinates, nor even the same (x,y) coordinates.

Output: The file output.txt which your program creates in the current directory should be formatted as follows:

First line: A space-separated list of safe location names that are along your solution path. This list should always start with the "start" location name and should always end with the "goal" location name.
If no solution was found (goal location was unreachable by the rover from the given starting point), write a single word **FAIL**.

For example, output.txt may contain:

```
start abcd efgh goal
```

Notes and hints:

- Please name your program "**homework.xxx**" where 'xxx' is the extension for the programming language you choose ("py" for python, "cpp" for C++17, and "java" for Java).
- Likely (but no guarantee) we will create 12 BFS, 19 UCS, and 19 A* test cases for grading.
- When you submit on vocareum, your program will run against the training test cases. Your program will be killed and the run will be aborted if it appears stuck on a given test case for more than **10 seconds**. During grading, the test cases will be of similar complexity, but you will be given up to 30 seconds per test case. Also note that vocareum has time limits for the whole set of 50 test cases, which are 300 seconds total for submission and 1800 seconds total for grading.
- You can tell vocareum to run only one test case instead of all of them, for faster debugging, by adding the following line anywhere in your code:
    - For python:
      # RUN_ONLY_TESTCASE x
    - For C++ or Java:
      // RUN_ONLY_TESTCASE x

  where x is a number from 1 to 50. Or you can write your own script to copy and run one test case at a time in the interactive shell of vocareum.
- The sample test cases are in **$ASNLIB/publicdata/** on vocareum. In addition to input.txt and output.txt, we also provide pathlen.txt with the total cost of the optimal path that is given in output.txt

- There is no limit on input size, number of safe locations, number of safe path segments, etc. other than specified above (32-bit integers, etc.). However, you can assume that all test cases used for grading will take < 30 secs to run on a regular laptop.
- If several optimal solutions exist, any of them will count as correct as long as its path length is within 0.1 of the optimal path length found by our reference algorithm and your path does not violate any energy/momentum rule.
- In vocareum, if you get **"child exited with value xxx"** on some test case, this means that your agent either crashed or timed out. Value 124 is for timeout. Value 137 if your program timed out and refused to close (and had to be more forcefully killed -9). Value 139 is for segmentation fault (trying to access memory that is not yours, e.g., past the end of an array). You can look up any other codes on the web.
- The 50 test cases given to you for training and debugging are quite big. We recommend that you first create your own small test cases to check for correct traversal order, correct loop detection, correct heuristic, etc. Also, there is no FAIL test case in the samples. To check for correct failure detection (no valid path exists from start to goal), you can take any of the test cases and either reduce the energy limit, or delete one arc that is on the solution path (you may have to try several times as other solution paths may still exist, though possibly more costly).
- Most of the 50 test cases given for training run in < 1 second with our reference code, with a few running in up to 3 seconds.