CSCI-561 - Fall 2019 - Foundations of Artificial Intelligence
Homework 1

Due September 26, 2019 23:59:59

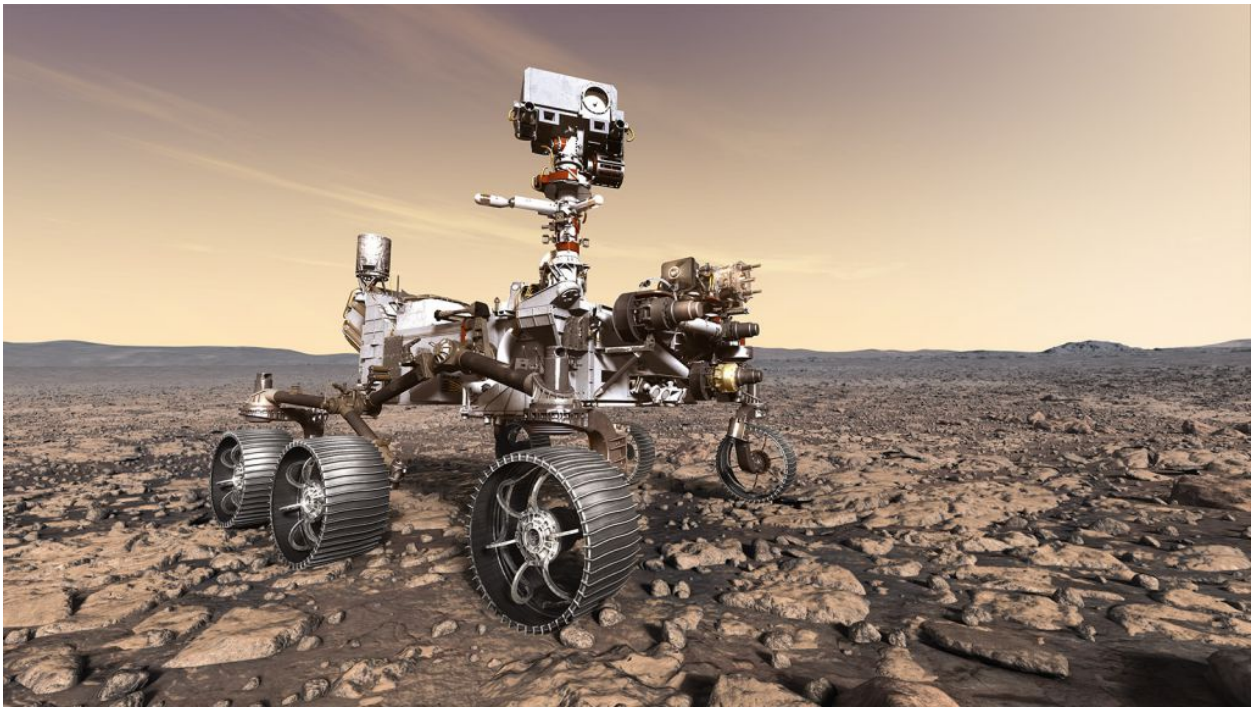

Image © NASA/JPL-Caltech

Guidelines

This is a programming assignment. You will be provided sample inputs and outputs (see below). Please understand that the goal of the samples is to check that you can correctly parse the problem definitions and generate a correctly formatted output. The samples are very simple and it should not be assumed that if your program works on the samples it will work on all test cases. There will be more complex test cases and it is your task to make sure that your program will work correctly on any valid input. You are encouraged to try your own test cases to check how your program would behave in some complex special case 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. You should upload and test your code on vocareum.com, and you will submit it there. You may use any of the programming languages provided by vocareum.com.

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 50 – N 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, **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 twist the problem of path planning a little bit just to give you the opportunity to deepen your understanding of search algorithms by modifying search techniques to fit the criteria of a realistic problem. To give you a realistic context for expanding your ideas about search algorithms, we invite you to take part in a Mars exploration mission. The goal of this mission is to send a sophisticated mobile lab to Mars to study the surface of the planet more closely. We are invited to develop an algorithm to find the optimal path for navigation of the rover based on a particular objective.

The input of our program includes a topographical map of the mission site, plus some information about intended landing site and target locations and some other quantities that control the quality of the solution. The surface of the planet can be imagined as a surface in a 3-dimensional space. A popular way to represent a surface in 3D space is using a mesh-grid with a Z value assigned to each cell that identifies the elevation of the planet at the location of the cell. At each cell, the rover can move to each of **8 possible neighbor cells**: North, North-East, East, South-East, South, South-West, West, and North-West. Actions are assumed to be deterministic and error-free (the rover will always end up at the intended neighbor cell).

The rover is not designed to climb across steep hills and thus moving to a neighboring cell which requires the rover to climb up or down a surface which is steeper than a particular threshold value is not allowed. This maximum slope (expressed as a **difference in Z elevation between adjacent cells**) will be given as an input along with the topographical map.

Search for the optimal paths

Our task is to move the rover from its landing site to one of the target sites for experiments and soil sampling. For an ideal rover that can cross every place, usually the shortest geometrical path is defined as the optimal path; however, since in this project we have some operational concerns, our objective is first to avoid steep areas and thus we want to minimize the path from A to B under those constraints. Thus, our goal is, roughly, finding the shortest path among the safe paths. What defines the safety of a path is the maximum slope between any two adjacent cells along that path.

Problem definition details

You will write a program that will take an input file that describes the terrain map, landing site, target sites, and characteristics of the robot. For each target site, you should find the optimal (shortest) safe path **from the landing site to that target**. A path is composed of a sequence of elementary moves. Each elementary move consists of moving the rover to one of its 8 neighbors. To find the solution you will use the following algorithms:

- Breadth-first search (BFS)
- Uniform-cost search (UCS)
- A* search (A*).

Your algorithm should return an **optimal path**, that is, with shortest possible operational path length. Operational path length is further described below and is not equal to geometric path length. If an optimal path cannot be found, your algorithm should return "FAIL" as further described below.

Terrain map

We assume a terrain map that is specified as follows:

A matrix with H rows (where H is a strictly positive integer) and W columns (W is also a strictly positive integer) will be given, with a Z elevation value (an integer number, to avoid rounding problems) specified in every cell of the WxH map. For example:

```
10  20  30
12  13  14
```

is a map with W=3 columns and H=2 rows, and each cell contains a Z value (in arbitrary units). By convention, we will use North (N), East (E), South (S), West (W) as shown above to describe motions from one cell to another. In the above example, Z elevation in the North West corner of the map is 10, and Z elevation in the South East corner is 14.

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 cell to any of its 8 neighbors counts for a unit path cost of 1. You do not need to worry about elevation differences (except that you still need to ensure that they are allowable and not too steep for your rover), or about the fact that moving diagonally (e.g., North-East) actually is a bit longer than moving along the North to South or East to West directions. So, any allowed move from one cell to an adjacent cell costs 1.

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

When running UCS, you should compute unit path costs in 2D. Assume that cells' center coordinates projected to the 2D ground plane are spaced by a 2D distance of 10 North-South and East-West. That is, a North or South or East or West move from a cell to one of its 4-connected neighbors incurs a unit path cost of 10, while a diagonal move to a neighbor incurs a unit path cost of **14** **as an approximation to $10\sqrt{2}$ when running UCS.**

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

When running A*, you should compute an approximate integer unit path cost of each move in 3D, by summing the horizontal move distance as in the UCS case (unit cost of 10 when moving North to South or East to West, and unit cost of 14 when moving diagonally), plus the absolute difference in elevation between the two cells. For example, moving diagonally from one cell with Z=20 to adjacent North-East cell with elevation Z=18 would cost 14+|20-18|=16. Moving from a cell with Z=-23 to adjacent cell to the West with Z=-30 would cost 10+|-23+30|=17. **You need to design an admissible heuristic for A* for this problem.**

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:	Two strictly positive 32-bit integers separated by one space character, for "W H" the number of columns (width) and rows (height), in cells, of the map.
Third line:	Two positive 32-bit integers separated by one space character, for "X Y" the coordinates (in cells) of the landing site. $0 \leq X \leq W-1$ and $0 \leq Y \leq H-1$ (that is, we use 0-based indexing into the map; X increases when moving East and Y increases when moving South; (0,0) is the North West corner of the map).
Fourth line:	Positive 32-bit integer number for the maximum difference in elevation between two adjacent cells which the rover can drive over. The difference in Z between two adjacent cells must be **smaller than or equal** ($\leq$) to this value for the rover to be able to travel from one cell to the other.
Fifth line:	Strictly positive 32-bit integer N, the number of target sites.
Next N lines:	Two positive 32-bit integers separated by one space character, for "X Y" the coordinates (in cells) of each target site. $0 \leq X \leq W-1$ and $0 \leq Y \leq H-1$ (that is, we again use 0-based indexing into the map).
Next H lines:	W 32-bit integer numbers separated by any numbers of spaces for the elevation (Z) values of each of the W cells in each row of the map.

For example:

```
A*
8 6
4 4
7
2
1 1
6 3
0  0  0  0  0  0  0  0
0 60 64 57 45 66 68  0
0 63 64 57 45 67 68  0
0 58 64 57 45 68 67  0
0 60 61 67 65 66 69  0
0  0  0  0  0  0  0  0
```

In this example, on a 8-cells-wide by 6-cells-high grid, we land at location (4, 4) highlighted in green above, where (0, 0) is the **North West corner** of the map. The maximum elevation change that the rover can handle is 7 (in arbitrary units which are the same as for the Z values of the map). We want to visit 2 targets, at locations (1, 1) and (6, 3), both highlighted in red above. The Z elevation map is then given as six lines in the file, with eight Z values in each line, separated by spaces.

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

N lines: Report the paths in the same order as the targets were given in the input.txt file. Write out one line per target. Each line should contain a sequence of X,Y pairs of coordinates of cells visited by the rover to travel from the landing site to the corresponding target site for that line. Only use a single comma and no space to separate X,Y and a single space to separate successive X,Y entries.
If no solution was found (target site unreachable by rover from given landing site), write a single word **FAIL** in the corresponding line.

For example, output.txt may contain:

```
4,4 3,4 2,3 2,2 1,1
4,4 5,4 6,3
```

Here the first line is a sequence of five X,Y locations which trace the path from the proposed landing site (4,4) to the first target (1,1). Note how both the landing site location and the target location are included in the path. The second line is a sequence of three X,Y locations which trace the path from the proposed landing site (4,4) to the second target (6,3).

The first path looks like this:

```
0  0  0  0  0  0  0  0
0 60 64 57 45 66 68  0
0 63 64 57 45 67 68  0
0 58 64 57 45 68 67  0
0 60 61 67 65 66 69  0
0  0  0  0  0  0  0  0
```

With the landing site shown in green, the target site in red, and each traversed cell in between in yellow. Note how one could have thought of a perhaps shorter path: `4,4 3,3 2,2 1,1` (straight diagonal from landing site to target site). But this was not possible for this rover as the move from 4,4 to 3,3 would incur a difference in Z of $|65 - 57| = 8$ which is too steep for this rover (difference must be ≤ 7 according to input.txt for this example).

And the second path looks like this:

```
0   0   0   0   0   0   0   0
0  60  64  57  45  66  68   0
0  63  64  57  45  67  68   0
0  58  64  57  45  68  67   0
0  60  61  67  65  66  69   0
0   0   0   0   0   0   0   0
```

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++, and "java" for Java). If you are using C++11, then the name of your file should be "homework11.cpp" and if you are using python3 then the name of your file should be "homework3.py".
- Likely (but no guarantee) we will create 15 BFS, 15 UCS, and 20 A* text cases.
- Your program will be killed after some time if it appears stuck on a given test case, to allow us to grade the whole class in a reasonable amount of time. We will make sure that the time limit for a given test case is at least 10x longer than it takes for the reference algorithm written by the TA to solve that test case correctly.
- There is no limit on input size, number of targets, etc other than specified above (32-bit integers, etc). If several optimal solutions exist, any of them will count as correct.

Extra credit:

Among the programs that get 100% correct on all 50 test cases,

- the fastest 10% on the A* test cases will get an extra 5% credit on this homework.

Example 1:

For this input.txt:

```
BFS
2 2
0 0
5
1
1 1
0 10
10 20
```

the only possible correct output.txt is:

```
FAIL
```

Example 2:

For this input.txt:

```
UCS
5 3
0 0
5
1
4 2
1 12 2 0 0
2 11 1 11 0
3 2 -1 9 0
```

one possible correct output.txt is:

```
0,0 0,1 1,2 2,1 3,0 4,1 4,2
```

Example 3:

For this input.txt:

```
A*
5 4
1 0
5
1
4 3
1 2 1 -2 0
1 1 1 2 9
9 -1 1 -1 11
1 2 1 1 -1
```

one possible correct output.txt is:

```
1,0 2,1 3,2 4,3
```