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

Due February 7, 2023 23:59:59

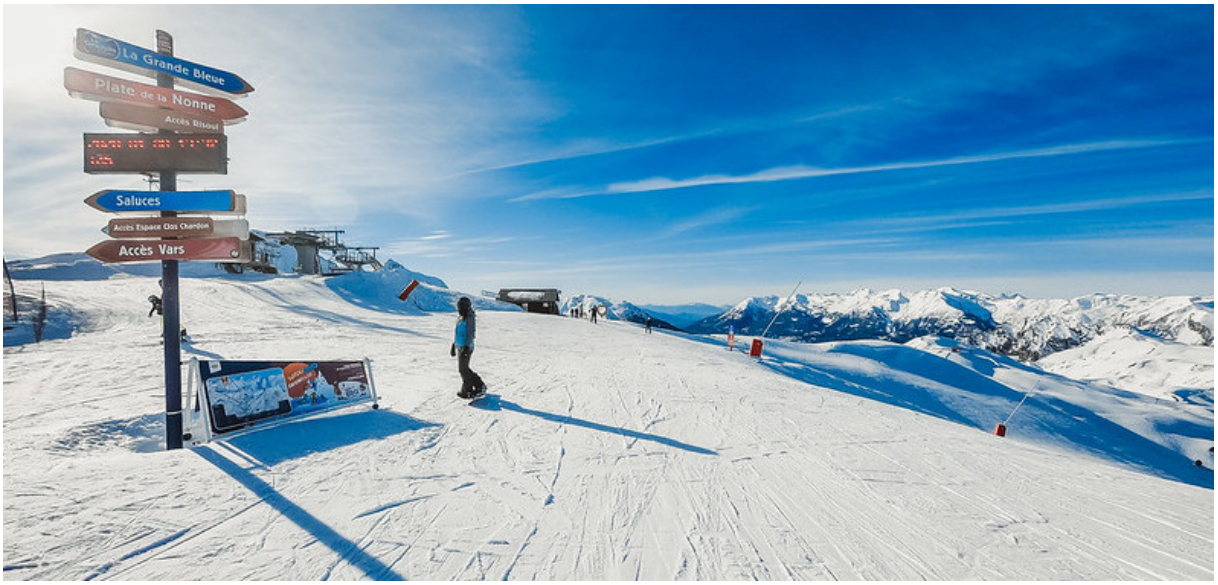

Image from flickr.com

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 below. You should upload and test your code on vocareum.com, and you will also 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 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, **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 to a Ski Resort. However, this is not a ski resort for the faint of heart. In this resort, there are no carved-out slopes and skiers must plan their own path to the lodge at the bottom of the mountain. The terrain is treacherous and the success of reaching the lodge depends on the level of the skier. We are invited to develop an algorithm to find the optimal path to reach our destination (and to a cup of hot cocoa) based on a particular objective.

The input of our program includes a topographical map of the mountain resort, plus some information about where our skier starts their journey, the position of the lodge and some other quantities that control the quality of the solution. The mountain can be imagined as a surface in a 3-dimensional space, and a popular way to represent it is by using a mesh-grid. The E value assigned to each cell represents the elevation of that location or whether it contains a tree we need to navigate around. At each cell, the skier can move to one 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 skier will always end up at the intended neighbor cell).

The skier cannot go over trees that are high enough to not have been covered with snow, nor go up a slope unless they have enough speed or stamina. Therefore, the value E in each cell can advise us on whether we can take that route (in case of tree and hills) or how much stamina moving into that cell will cost the skier if they move into it.

Search for the optimal paths

Our task is to lead the skier from their start position to the ski lodge, where they can celebrate a good ski day with a cup of hot chocolate. If we had a very advanced skier that can go across any land without a problem, usually the shortest geometrical path is defined as the optimal path; however, since our skiers might be at different levels (we will define this as stamina), our objective is to avoid high trees as well as very steep areas, unless we have gained some momentum first. Thus, we want to minimize the path from A to B under those constraints. Our goal is, roughly, finding the shortest path among the safe paths. What defines the safety of a path is whether there are trees we can't cross and the elevation of the cells along that path.

Problem definition details

You will write a program that will take an input file that describes the land, the starting point, potential lodges we can relax in, and some other characteristics for our skier. For each lodge location, you should find the optimal (shortest) safe path **from the starting point to that target lodge**. A path is composed of a sequence of elementary moves. Each elementary move consists

of moving the skier from their current position 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 journey cost. Journey cost 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 for the mountain 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 value E (an integer number, to avoid rounding problems) specified in every cell of the WxH map. If E is a negative integer, this means there is a tree of height |E| in that cell. If E is a positive integer, the value represents the elevation of that cell. For example:

```
10  20  -30
12  13  40
```

is a map with W=3 columns and H=2 rows, and each cell contains an E 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, elevation E in the North-West (NW) corner of the map is 10, and it is 40 in the South-East (SE) corner, which means our skier will spend more stamina to move into the SE corner than the NW corner.

Every skier will have a stamina value S that determines whether they can climb up certain elevations. **A move from E = 20 to E = 40 is only allowed if a skier's stamina is higher than or equal to 40 – 20 = 20. <u>Important Note:</u> Stamina remains as a constant value.**

Notice that there is a tree of height 30 in the NE corner. If a cell contains a tree, there are a few factors that determine whether a move into that cell is allowed:

- **If your current E is higher than or equal to tree height |E|, you are allowed to move into the cell**. In this case, we imagine the tree is covered with snow of height |E|. Therefore, if our tree is height 30 as above, it would be allowed to move into that cell from the cell with elevation 40, but not from others.
- **If your current E is lower than the tree height |E|, a move into that cell is not allowed from your current cell.** Note that your stamina (or momentum) does not change whether a move like this is allowed.

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 the elevation levels or about the fact that moving diagonally (e.g., North-East) is actually a bit longer than moving along the North to South or East to West directions. **However, you still need to make sure the move is allowed by checking how steep the move is (depends on the skier's stamina) or whether a tree is involved. Therefore, 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. You still need to make sure the move is allowed, in the same way you did for BFS.**

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

When running A*, you will have modified rules for allowed moves AND you should compute an approximate integer unit path cost for each move according to how the elevation of the terrain changes. We will explain these rules using elevations E_{curr} (current cell elevation), E_{next} (potential next cell elevation), and E_{prev} (previous cell elevation), as well as stamina S (given as input) and momentum M (defined below).

Allowed moves for A*: For running A*, we modify whether a move is allowed by considering momentum (M), that is whether we gained some speed by going down in elevation in our most recent move. If we are currently at a cell with E_{curr}, whether we are allowed to go into a cell E_{next} is determined by whether our momentum M going from E_{prev} to E_{curr} can assist us. We will define M as follows:

$$\mathrm{M} = \begin{cases} \max(0, E_{prev} - E_{curr}), & (E_{next} - E_{curr}) > 0 \\ 0, & (E_{next} - E_{curr}) \leq 0 \end{cases}$$

That is, when we are going down from E_{prev} to E_{curr} and then up from E_{curr} to E_{next}, momentum M is > 0 and will possibly assist us in reaching cells with higher E_{next} elevations. If the next move is going up in elevation ($E_{next} > E_{curr}$), a move will only be allowed if ($E_{next} <= E_{curr} + S + M$). For BFS/UCS, this rule was ($E_{next} <= E_{curr} + S$). M = 0 initially at the starting position. Note how in all cases other than going down from E_{prev} to E_{curr} and then up from E_{curr} to E_{next}, momentum M is 0 according to the above definition. **Note that momentum does not accumulate across multiple moves. Only the latest E_{prev} to E_{curr} are considered when computing M.**

If a tree is involved, the rules to determine whether a move is allowed do not change from the BFS/UCS cases. If you are allowed to move into a cell with a tree, it acts as land with elevation |E| from then on.

Path cost for A*: You should also compute an approximate integer unit path cost for each move for A*, which is now approximately 3D. The cost of a move is computed by considering both 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) and the change in elevation levels of the land. The cost C for the move is hence defined as follows:

C = (Horizontal Move Distance) + (Elevation Change Cost)

Where the Elevation Change Cost is defined as follows:

$$\text{Elevation Change Cost } = \begin{cases} 0, & (E_{next} - E_{curr}) \leq M \\ max(0, E_{next} - E_{curr} - M), & (E_{next} - E_{curr}) > M \end{cases}$$

Thus, intuitively, the elevation change cost is how much we are going uphill, possibly minus how much momentum we have from going downhill on the previous move.

Examples:

- If our previous cell was E_{prev} = 20 and our current cell is E_{curr} = 8, we have M = 12. If our stamina is 30, we are allowed to go into a cell with E_{next} up to E_{curr} + M + S = 8 + 12 + 30 = 50. Assume we choose a cell with E_{next} = 25. If we're moving diagonally, our path cost becomes: 14 (Move Distance) + (25 – 8 – 12 = 5) (Elevation Change Cost) = 19.

- If our previous cell was E_{prev} = 20 and our current cell is E_{curr} = 25, we have M = 0. If our stamina is 15, we are allowed to go into, for example, a cell with E_{next} = 35 (since 35 – 25 < 0 + 15). If we're moving South, our path cost becomes: 10 (Move Distance) + (35 – 25) (Elevation Change Cost) = 20.

- If our current cell is E_{curr} = 40 and our next cell is E_{next} = 15, the move is allowed since we are going downhill. If we're moving East, our path cost becomes: 10 (Move Distance) + (0) (Elevation Change Cost) = 10.

- If our previous cell was E_{prev} = 12 and our current cell is E_{curr} = -5, we have M = 7, since we treat tree cells as |E| elevation once we move into them. If our stamina is 5, we can go into, for example, a cell with E_{next} = 30 (since 30 – 12 < 7 + 5). If we're moving SW, our path cost becomes: 14 (Move Distance) + (30 – 5 – 7) (Elevation Change Cost) = 32.

Remember: In addition to computing the path cost, you also 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 starting position for our skier. $0 \leq X \leq W-1$ 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). **Starting point remains the same for each of the N lodge sites below and will never contain a tree.**

Fourth line: Positive 32-bit integer number for the stamina S of the skier which determines how advanced our skier is. S will be used to compute allowed moves if we're moving into a non-tree cell.

Fifth line: Strictly positive 32-bit integer N, the number of lodges on the mountain.

Next N lines: Two positive 32-bit integers separated by one space character, for "X Y" the coordinates (in cells) of each lodge 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). **These N target lodge sites are not related to each other, so you will run your search algorithm on each lodge site and write the result to the output as specified below. We will never give you a lodge site that is the same as the starting point. They will never contain a tree.**

Next H lines: W 32-bit integer numbers separated by any numbers of spaces for the M values of each of the W cells in each row of the map. Each number can represent the following cases:

- E >= 0, snowy mountain slope with elevation E
- E < 0, tree of height |E| that might be covered with snow depending on the elevation we approach it from

For example:

```
A*
8 6
4 4
5
2
2 1
6 3
-10 40 34 21 42 37 18  7
-20 10  5 27 -6  5  2  0
-30  8 17 -3 -4 -1  0  4
-25 -4 12 14 -1  9  6  9
-15 -9 46  6 25 11 31 -21
-5  -6 -3 -7  0 25 53 -42
```

In this example, on an 8-cells-wide by 6-cells-high grid, we start at location (4, 4) highlighted in green above, where (0, 0) is the **North West corner** of the map. The maximum stamina that the skier has is 5 (in arbitrary units which are the same as for the E values of the map). We have 2 possible lodge sites, at locations (2, 1) and (6, 3), both highlighted in red above. The map of the land is then given as six lines in the file, with eight E values in each line, separated by spaces. The negative values are trees.

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 lodge sites were given in the input.txt file. Write out one line per target lodge. Each line should contain a sequence of X,Y pairs of coordinates of cells visited by the skier to travel from the starting point to the corresponding lodge 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 (lodge was unreachable by the skier from the given starting point), write a single word **FAIL** in the corresponding line. Our skier needs a rescue in this case. ☺

For example, output.txt may contain:

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

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

The first path looks like this:

```
-10  40  34  21  42  37  18   7
-20  10   5  27  -6   5   2   0
-30   8  17  -3  -4  -1   0   4
-25  -4  12  14  -1   9   6   9
-15  -9  46   6  25  11  31 -21
-5   -6  -3  -7   0  25  53 -42
```

With the starting point shown in green, the lodge sites in red, and each traversed cell in between in yellow. Note how one could have thought of a perhaps shorter path: `4,4 4,3 3,2 2,1` (general lower |E| values). But this was not allowed, since once we get to position `4,3` we are blocked by the tree in `3,2`.

And the second path looks like this:

```
-10  40  34  21  42  37  18   7
-20  10   5  27  -6   5   2   0
-30   8  17  -3  -4  -1   0   4
-25  -4  12  14  -1   9   6   9
-15  -9  46   6  25  11  31 -21
-5   -6  -3  -7   0  25  53 -42
```

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++11, and "java" for Java).
- 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 target lodges, etc. other than specified above (32-bit integers, etc.). However, you can assume that all test cases will take < 30 secs to run on a regular laptop.
- If several optimal solutions exist, any of them will count as correct.
- Actual test cases used for grading will be significantly more complex than the 3 examples shown below (e.g., that could have 500x500 maps or bigger). The examples below are mostly to make sure you can correctly parse the inputs and produce correctly formatted outputs.

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 1
1 5  1  -1  -2
6 2  4   10  3
9 8 -10 -20  40
```

one possible correct output.txt is:

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

Example 3:

For this input.txt:

```
A*
5 4
0 1
3
1
4 3
20  2 1 -2 -10
-8  1 10 2 -20
9  -1 4  15 11
6  -5 1  1 -1
```

one possible correct output.txt is:

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