CSCI-561 - Spring 2021 - Foundations of Artificial Intelligence
Homework 1

Due February 22nd, 2021 23:59:59

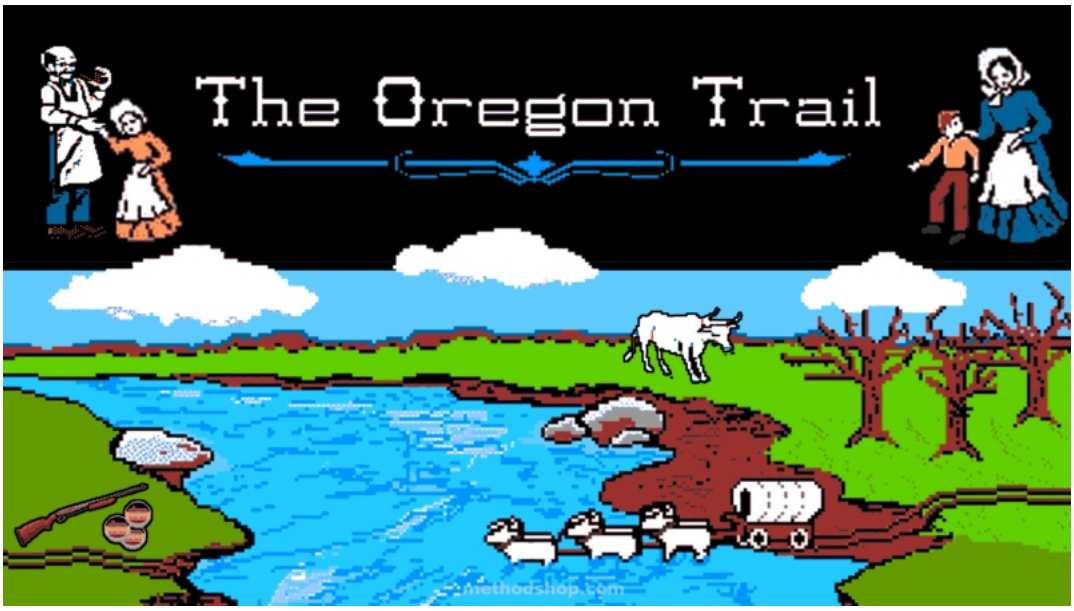

Image © The Original Oregon Trail Game

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 – 4xN points where N is the number of failed test cases, or 0 if your code fails more than 25 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 context for expanding your ideas about search algorithms, we invite you to take part in a simplified version of the computer game *The Oregon Trail*. In this educational computer game, the player takes the role of leader to a group of settlers in 1848, traveling with a wagon from Independence, Missouri, to Oregon's Willamette Valley. The goal is to reach Oregon without running out of provisions or falling victim to nature. We are invited to develop an algorithm to find the optimal path for navigation of the wagon based on a particular objective.

The input of our program includes a topographical map of the land, plus some information about where our party starts their journey, the intended site our party wants to settle and some other quantities that control the quality of the solution. The land 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 M value assigned to each cell will represent how muddy the patch of land is or whether it contains a rock. At each cell, the wagon 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 wagon will always end up at the intended neighbor cell).

The wagon cannot go over rocks that are too high, and the wheels are such that, as the land gets muddier, the wagon slows down. Therefore, the value M in each cell can advise us on whether we can take that route (in case of rocks) or how much moving into that cell will cost the settler party in terms of time if they move into it (in case of mud).

Search for the optimal paths

Our task is to lead the party of settlers from their start position to the land they aim to reach. If we had the ideal vehicle that can go across any land without a slow-down, usually the shortest geometrical path is defined as the optimal path; however, since our wagon is far from ideal, our objective is to avoid rocks we can't cross over as well as really muddy areas. 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 rocks we can't cross and the muddiness 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 settling sites for our party of settlers, and some other characteristics for our wagon. For each settling site, you should find the optimal (shortest) safe path **from the starting point to that target site**. A path is composed of a sequence of elementary moves. Each elementary move consists of moving the wagon party 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 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 value M (an integer number, to avoid rounding problems) specified in every cell of the WxH map. If M is a negative integer, this means there is a rock of height |M| in that cell. If M is a positive integer, the value represents the level of muddiness 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 M 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, mud level M in the North West corner of the map is 10, and it is 40 in the South East corner, which means our wagon will take more time to move into the SE corner than the NW corner. There is a rock of height 30 in the NE corner.

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 muddiness levels 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, but **you still need to make sure the move is allowed by checking how steep the move is. 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 if a cell with a rock is involved.**

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

When running A*, you should compute an approximate integer unit path cost of each move by also considering the muddiness levels of the land, 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 muddiness level in the cell we are trying to move in to, plus the absolute height we have to traverse from our current cell to the next (cells where M >= 0 have height 0). For example, moving diagonally from the current position with M=-2 to an adjacent North-East cell with mud level M=18 would cost 14 (diagonal move) + 18 (mud level) + |0-2| (height change) = 34. Moving from a cell with M=1 to an adjacent cell with M=5 to the West would cost 10+5+0=15. **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 land 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 party. $0 \leq X \leq W-1$ 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 target sites below.**

Fourth line: Positive 32-bit integer number for the maximum rock height that the wagon can climb between two cells. The difference of heights between two adjacent cells must be **smaller than or equal** ($\leq$) to this value for the wagon to be able to travel from one cell to the other.

Fifth line: Strictly positive 32-bit integer N, the number of settling sites.

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

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:

- M >= 0, muddy land with height 0 and mud-level M
- M < 0, rock of height |M| with mud-level 0

For example:

```
A*
8 6
4 4
3
2
1 1
6 3
-10 40 34 21 42 37 18  7
-20 10  6 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  5 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 elevation change that the wagon can travel is 3 (in arbitrary units which are the same as for the M values of the map). We have 2 possible targets sites, at locations (1, 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 M 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 target sites 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 wagon party to travel from the starting point to the corresponding settling 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 (settling site unreachable by the wagon from the given starting point), write a single word **FAIL** in the corresponding line.

For example, output.txt may contain:

```
4,4 4,3 3,2 2,1 1,1
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   6  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   5  11  31 -21
-5   -6  -3  -7   0  25  53 -42
```

With the starting point shown in green, the settling sites 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 as good as this path has much higher mud levels and therefore costs more for A*.

And the second path looks like this:

```
-10  40  34  21  42  37  18   7
-20  10   6  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   5  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++, 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.). 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.

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
1 0
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:

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