CSCI 1100 — Computer Science 1 Homework 5
Lists of Lists; Grids; Path Planning

Overview

This homework is worth **100 points** total toward your overall homework grade. It is due Thursday, March 24, 2022 at 11:59:59 pm. As usual, there will be a mix of autograded points, instructor test case points, and TA graded points. There are two parts to the homework, each to be submitted separately. Both parts should be submitted by the deadline or your program will be considered late.

See the handout for Submission Guidelines and Collaboration Policy for a discussion on grading and on what is considered excessive collaboration. These rules will be in force for the rest of the semester.

You will need the data files we provide in `hw5_files.zip`, so be sure to download this file from the `Course Materials` section of Submitty and unzip it into your directory for HW 5. The zip file contains utility code, data files and example input / output for your program.

Overview

Many problems in computer science and in engineering are solved on a two dimensional numerical grid using techniques that are variously called "gradient ascent" (or "descent"), greedy search, or hill-climbing. We are going to study a simplified version of this using hand-generated elevation data.

The main representation we need is a list of lists of "heights" (also called "elevations", but we will use the simpler term "heights" here). For example,

```
grid = [[15, 16, 18, 19, 12, 11],
        [13, 19, 23, 21, 16, 12],
        [12, 15, 17, 19, 22, 10],
        [10, 14, 16, 13,  9,  6]]
```

has four lists of six integer entries each. Each entry represents an height — e.g. meters above sea level — and the heights are measured at regularly-spaced intervals, which could be as small as centimeters or as large as kilometers. The USGS, United States Geological Survey, maintains and distributes elevation data like this, but private companies do so as well. Such data are important for evaluating water run-off, determining placement of wind turbines, and planning roads and construction, just to name a few uses. We are going to use the analogy of planning hiking paths on a plot of land.

Questions we might ask about this data include,

1. What is the highest point (greatest height)? This is also called the "global maximum" because it is the greatest value in the data. In our example, this height value is 23 and it occurs in list 1, entry 2. We will refer to this as row 1, column 2 (or "col 2"), and write these values as a tuple, (1, 2), where we assume the first value is the row and the second is the column. We refer to (1, 2) as a "location".

2. Are there "local maxima" in the data? These are entries whose value is greater than their

immediately surrounding values, but smaller than the global maximum. In our example there is a local maxima of 22 at location (2, 4).

3. Starting at a given location, what is the best path to the global maxima? This is a tricky question because we need to define "best path". Here is a simple one: can we start at a given location and only take the steepest route and get to the global maximum (can we hike up to the "peak")? For example, if we start at (3, 0) then the path through locations (3, 0), (3, 1), (3, 2), (2, 2), (1, 2) follows the steepest route and reaches the top. This is a "gradient ascent" method. But, if we start at location (3, 5) then we will generate the route (3, 5), (3, 4), (2, 4), but then there is no way to continue up to reach the global maximum.

There are many, many more questions that we can ask and answer. Some of them can be solved easily, while others require sophisticated algorithms and expensive computations.

Before getting started on the actual assignment, it is important to define the notion of a "neighbor" location in the grid — one that we are allowed to step to from a given location. For our purposes, from location `(r, c)`, the neighbor locations are `(r-1, c)`, `(r, c-1)`, `(r, c+1)`, and `(r+1, c)`. In other words, a neighbor location must be must be in the same row or the same column as the current location. Finally, neighbors can not be outside the grid, so for example at location `(r, 0)` only `(r-1, 0)`, `(r, 1)`, `(r+1, 0)` are allowed as neighbors,

Getting Started

Please download `hw5_files.zip` and place all the files in the same folder that you are going to write your solutions. Files `hw5_util.py` and `hw5_grids.txt` are quite important: `hw5_util.py` contains utility functions to read grids and starting locations from `hw5_grids.txt`. In particular,

- `hw5_util.num_grids()` returns the number of different grids in the data,
- `hw5_util.get_grid(n)` returns grid `n`, where `n==1` is the first grid and `n == hw5_util.num_grids()` is the last.
- `hw5_util.get_start_locations(n)` returns a list of tuples giving one or more starting locations to consider for grid `n`,
- `hw5_util.get_path(n)` returns a list of tuples giving a possible path for grid `n`.

We suggest you start by playing around with these functions and printing out what you get so that you are sure you understand.

You may assume the following about the data:

1. The grid has at least two rows.
2. Each row has at least two entries (columns) and each row has the same number of columns.
3. All heights are positive integers.
4. The start locations are all within the range of rows and columns of the grid.
5. The locations on the path are all within the range of rows and columns of the grid.

Part 1

Write a python program, `hw5_part1.py` that does the following:

1. Asks the user for a grid number and loops until one in the proper range is provided. Denote the grid number as `n`.

2. Gets grid `n`

3. Asks the user if they want to print the grid. A single character response of `'Y'` or `'y'` should cause the grid to be printed. For anything else the grid should not be printed. When printing, you may assume that the elevations are less than 1,000 meters. See the example output.

4. Gets the start locations associated with grid `n` and for each it prints the set of neighbor locations that are within the boundaries of the grid. For example if grid `n` has 8 rows and 10 columns, and the list of start locations is

   ```
   [(4, 6), (0, 3), (7, 9)]
   ```

   then the output should be

   ```
   Neighbors of (4, 6): (3, 6) (4, 5) (4, 7) (5, 6)
   Neighbors of (0, 3): (0, 2) (0, 4) (1, 3)
   Neighbors of (7, 9): (6, 9) (7, 8)
   ```

   Very important: we strongly urge you to write a function called `get_nbrs` that takes as parameters a row, col location, together with the number of rows and columns in the grid, and returns a list of tuples containing the locations that are neighbors of the row, col location and are within the bounds of the grid. You will make use of this function frequently.

5. Gets the suggested path, decides if it is a valid path (each location is a neighbor of the next), and then calculates the total downward elevation change and the total upward elevation change. For example using the grid above, if the path is

   ```
   (3, 1), (3, 0), (2, 0), (1, 0), (1, 1), (0, 1), (0, 2), (1, 2)
   ```

   the downward elevation changes are from (3, 1) to (3, 0) (change of 4) and from (1, 1) to (0, 1) (change) of 3 for a total of 7, and the upward elevation changes are from (3, 0) to (2, 0), from (2, 0) to (1, 0), from (1, 0) to (1, 1), from (0, 1) to (0, 2) and from (0, 2) to (1, 2) for a total of $(2 + 1 + 6 + 2 + 5) = 16$). The output should be:

   ```
   Valid path
   Downward 7
   Upward 16
   ```

   If the path is invalid, the code should print

   ```
   Path: invalid step from point1 to point2.
   ```

   Here `point1` and `point2` are the tuples representing the start and end of an invalid step.

Submit just the file `hw5_part1.py` and nothing else.

Part 2

Revise your solution to Part 1 and submit it as `hw5_part2.py`. The program should again ask the user for the grid number, but it should not print the grid. Next, it should find and output the location and height of the global maximum height. For example for the simple example grid, the output should be

```
global max: (1, 2) 23
```

You may assume without checking that the global maximum is unique.

The main task of Part 2 is to find and output two paths from each start location for the grid. The first is the steepest path up, and the second is the most gradual path up. The steps on each path must be between neighboring locations as in Part 1. Also, on each path no steps to a location at the same height or lower are allowed, and the step size (the change in height) can be no more than a maximum step height (difference between heights at the new location and at the current location). Your code must ask the user for the value of this maximum step height.

Next, determine for each path if it reaches the location of the global maximum height in the grid, a local maximum, or neither. The latter can happen at a location where the only upward steps are too high relative to the height at the current location. Of course, true hiking paths can go both up and down, but finding an "optimal path" in this more complicated situation requires much more sophisticated algorithms than we are ready to develop here.

As an example of the required results, here is the same grid as above:

```
grid = [[15, 16, 18, 19, 12, 11],
        [13, 19, 23, 21, 16, 12],
        [12, 15, 17, 19, 20, 10],
        [10, 14, 16, 13,  9,  6]]
```

starting at location (3, 0) with a maximum height change of 4, the steepest path is (3, 0), (3, 1), (3, 2), (2, 2), (2, 3), (1, 3), (1, 2), while the most gradual path is (3, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 2). Both reach the global maximum, and both avoid stepping to the global maximum the first time they are close because the step height is too large. Note that both the steepest and most gradual paths from location (3, 5) would end at the local maximum (2, 4). The steepest path would end after four steps (five locations on the path) and the most gradual would end after six steps (seven locations on the path). If the max step height were only 3, then both paths from (3, 5) would stop at location(3, 4) before any maximum is reached.

Paths should be output with 5 locations per line, for example

```
steepest path
(3, 0) (2, 0) (1, 0) (0, 0) (0, 1)
(0, 2) (0, 3) (1, 3) (1, 2)
global maximum
```

See the example output for further details.

Finally, if requested by the user, output a grid — we'll call it the "path grid" — giving at each location the number of paths that include that location. This can be handled by forming a new list of lists, where each entry represents a count — initialized to 0. For each path and for each location `(i, j)` on the path, the appropriate count in the list of lists should be incremented. At the end,

after all paths have been generated and added to the counts, output the grid. In this output, rather than printing a 0 for a locations that are not on any path, please output a `'.'`; this will make the output clearer. See the example output.

Notes

1. In deciding the choice of next locations on a path, if there is a tie, then pick the one that is earlier in the list produced by your `get_nbrs` function. For example starting at (0, 5) with elevation 11 in the above example grid, both (0, 4) and (1, 5) have elevation 12. In this case (0, 4) would be earlier in the `get_nbrs` list and therefore chosen as the next location on the path.

2. Please do not work on the path grid output — the last step — until you are sure you have everything else working.

3. Both the most gradual and steepest paths are examples of *greedy* algorithms where the best choice available is made at every step and never reconsidered. More sophisticated algorithms would consider some form of backtracking where decisions are undone and alternatives reconsidered.