The Mountain

Victor Cionca and Alison O'Shea

Description

Your program wakes up at the bottom of a mountain. The whole area is shrouded in deep fog so you only have visibility at a short distance around. Your goal is to get to the top of the mountain in as little time as possible.

The landscape is implemented as a 2D matrix on top of which a mountain of a given height and with some irregularities (plateaus) has been placed. The mountain has a single peak.

Each element of the 2D matrix contains a value that represents the altitude (height) of that cell, with the mountain peak representing the highest point in the lanscape, or the maximum matrix value.

You will not have access to the 2D matrix, or the matrix's dimensions (height and width). The only thing you can do is make calls to the function

```
void generate_view(float view[VIEW_SIZE][VIEW_SIZE], int center_y, int center_x)
```

that will populate a 2D *view* array from the landscape matrix centered on the `center_y` and `center_x` coordinates (resp row and column). Every time you call this function the program will increment your call count with the goal being to implement a search solution that makes the least calls to `generate_view`.

When you believe that you have found the peak at a position `x,y` you must call the function `int declare_peak(x,y)`. The function will return `1 (true)` if the peak is correctly identified or `0 (false)` otherwise. You can call this function any number of times, however every time it is called it will increment your call count and this will add up to your total.

NOTE The `generate_view` function will stop working after it's been called as many times as the number of cells in the landscape matrix. Same for `declare_peak` which will only return 0 after that point.

This is effectively a search problem and there are various possible solutions:

- brute force searching through the matrix – although you don't know the matrix dimensions (number of rows and columns) so you must perform some exploration at first
- random search
- gradient-based hill climbing.

NOTE Be careful with x and y coordinates. The y coordinates represent the row index in a 2D matrix, while x represents columns.

Tasks

You are provided with a C source file `gradient.c` containing code for

- landscape generation
- evaluation of your solution.

You are also provided with a C header file `gradient.h` containing some C macros, variable type definitions and function prototypes.

In a separate C source file (e.g. `gradient_sol.c`) implement the function `path_point find_highest_point()`.

This function should start at a certain coordinate in the matrix (e.g. x=0, y=0) and explore the 2D landscape matrix using the `generate_view` function. Eventually when you think you have found the peak you should call `declare_peak` as instructed above. The function must work with a local 2D view matrix which must have `VIEW_SIZE` rows and columns. `VIEW_SIZE` is a macro defined in the program and you can use it as is.

Your function should return a variable of type `path_point` which is a `struct` containing two integer fields, `x` and `y`. For example, if you found the peak and store it within your function in variables `peak_x` and `peak_y` you should return it with the following code:

```
path_point ret;
ret.x = peak_x;
ret.y = peak_y;
return ret;
```

When your function returns, the total number of lookups into the landscape matrix will be evaluated.

NOTE In your solution C source file you must include `"gradient.h"`.

You should only submit the solution file.

Compiling the code

The landscape generation code makes use of mathematical functions which require the `math` library. This must be included during compilation.

Note that your application will consist of two source files:

- the main file `gradient.c` with landscape generation and main function
- your solution file (e.g. `gradient_sol.c`).

If using an IDE you have to select both files for compilation.

If using `gcc` you can compile your code with: `gcc gradient.c gradient_sol.c -lm -o gradient` to generate the `gradient` executable.

Evaluating your own solution

You should try to design your solution so that it finds the peak with the least amount of calls to `generate_view()` and `declare_peak()`. You can evaluate your solution in two ways:

- single run, using the `single_run(int seed)` function
  - `seed` is a value that will be used to seed the random number generator; using the same seed will result in the same configuration of the landscape matrix
  - if the seed is set to `-1` the current system time will be used instead which means the landscape matrix will have a different configuration every time
  - the single run is useful for debugging your solution; see below for more instructions
- performance evaluation, using the `performance_eval()` function; this will run your function against 100 randomly generated landscapes
  - it will print for each one the index of the instance, the true location of the peak as well as the number of calls that were made.

Debugging

It is recommended that you start debugging with a single run and once you are able to find the peak, try the performance evaluation. This will identify your overall performance as well as whether there are cases where your solution fails to find the peak (as indicated above, if you don't find the true peak you will get maximum amount of calls). If there are such cases you can replicate them with `single_run` by using as `seed` the index of the instance.

Example output from `performance_eval`:

```
Attempt 26 peak at 82-130, found in 33 tries
Attempt 27 peak at 82-114, found in 19890 tries
```

Shows that with landscape instance 26 the peak was found in 33 calls, whereas in 27 the maximum number of tries is reported which means the peak was not found correctly.

This can then be inspected further using the following code:

```
single_run(27); // Run only for the instance that caused problems
print_matrix(); // print matrix for inspection
for (i=0;i<queries_made;i++)        // print the path taken
    printf("%d-%d\n", user_path[i].y, user_path[i].x);
```

- in the above example you should comment out the `free_landscape();` call within `single_run` so you can access the matrix after the function completes
- the `print_matrix` function prints the entire landscape matrix
- you have access to the number of queries that you made to the landscape through the variable `queries_made`
- the program stores the path that you have taken through the matrix in the `user_path` array of `path_point` variables; the array should have `queries_made` elements and you can print them with a `for` loop
- more debugging information can be obtained by commenting out the `printf` statement inside the `add_noise` function, which generates the "plateaus".

Competitive evaluation

You have the option of submitting your solution to competitive evaluation. A website will be provided where you will be able to submit your code and this will be tested on a server with the score published on a leaderboard.

This evaluation will be different than the local evaluation that you can perform on your own with the provided code:

- you will not have access to the landscape matrix variable or any other variables that you can find in the provided source file
- you will not be able to obtain the dimensions of the matrix
- you can only use the `generate_view` function to obtain information from the matrix.

More information on this will be made available soon.

Grading

Brute-force and random attempts are capped at 20/35 points.

Further points are granted based on the score obtained in the performance evaluation.

Basic grading 20/35

- view array: correct definition 2
- calling `generate_view` correctly 2
- correct scanning of the 2D view array 3
- looking for the peak within the view 3
- calculating the coordinates of the next view point 4
  - this requires translating from view-coordinates to landscape coordinates
  - for those that perform brute-force or random search (e.g. moving the view point in a regular pattern or a random pattern) only 2 points awarded
- returning correct value 2
  - declaring the `path_point` variable
  - setting its fields
- finding the peak at least once during the `performance_eval` 4
  - solutions that rely on reading the global variables from the provided code, such as the true location of the peak, dimensions of the matrix, etc will not be awarded these points.