CITS2200 Project 2020

Daniel Marwick
CITS2200 - Data Structures and Algorithms
University of Western Australia

May 28, 2020

Part 1 - Flood Fill Count

Explanation

Here I simulated an actual iterative flood fill, using a stack and hash-set from the Java standard library. The set is to keep track of which cells have been counted already, and the stack keeps track of the current "path".

Correctness

To store the position of a pixel, I have used a method that allows 2 numbers to be represented by just one integer. A point (y, x) would be translated as follows:

```
array[y][x] -> y * width + x
```

such that `width` (represented as `cols` in the code) represents the width of the image, or number of columns. This allows for each pixel on the image to be assigned to a unique integer. To move left or right, we add or subtract 1 and to move up or down, we add or subtract `cols`. The x- and y- coordinates can be pretty easily extracted from this too, using the following logic:

```
x = position % cols
y = position / cols
```

Every iteration, the code tries looking in every direction in the order left, down, right, up. When it can't move in any direction, due to being at an edge of the contiguous colour region and/or all neighbours being covered already, we backtrack by popping the current point off the stack. This process ensures that every single pixel in the region of the same colour gets visited.

Complexity

In the worst case, every pixel has to be visited making this an $O(P)$ algorithm. Every time a new pixel is added to the count, maintaining a set recording every visited pixel ensures that there are no repeats. Additionally, checking the set in every iteration does not add complexity as the HashSet data structure from the Java standard library is a hash-based implementation that allows basic operations like adding and checking to be done in constant time.

Part 2 - Brightest Square

Explanation

This algorithm was inspired by an answer on *stackoverflow.com* which was to explain how to obtain the maximum sum of a subsection of a 2-dimensional array of integers faster than one would using the most naive $O(Pk^2)$ solution. It involved recording the cumulative brightness of every pixel from the top-left corner to itself in another array in $O(P)$ time so that the solution could be found without too many nested loops.

Correctness

The trade-off to achieving this time complexity is requiring more space. Three new arrays had to be made, with the same size as the image. One was `totals` which as mentioned, at every index contained the brightness of the rectangular region spanning from position $(0, 0)$ to (x, y). The other two were tracking the running sum of each horizontal strip in `hTotals` and each vertical strip in `vTotals`. These two were used to fill in `totals`. The procedure for generating this data was straightforward enough, `vTotals` and `hTotals` were able to be done in one set of 2 nested for-loops. For the vertical strips, if the y-coordinate was zero then that value would just be the image pixel value at that point. Directly below (y increasing), would be the image pixel value at that point plus the strip total just before it. Similar logic applies to the horizontal strip array, and with some perhaps questionable uses of if-statements and `continue`, both arrays could be processed in one set of 2 nested for-loops. Now, filling the `totals` array was doable in a similar way. along the left and upper edges, the cumulative brightness could be copied directly from the corresponding strip-total array and once those were set, each pixel's cumulative brightness value could be derived from the strip arrays and previous calculations in the same array. Once this is done, getting the brightest square of size k can be done with a bit of logic.
As is made clear Figure 1, the square-sum at any point can be calculated by subtracting 2 rectangular sums and re-adding the top left corner sum. If the image is already in the top left corner, the sum is trivially the cumulative sum at (k, k). If at only one of the top or left edges, one of the subtracted rectangles does not need to be considered. With this, any k by k square can be calculated and thus the brightest one can be found.

Complexity

While possibly mentally taxing to code two separate loops for processing the image before actually obtaining the answer (and also as mentioned before, requiring a bit more space), the benefit is being able to compute the brightest square in $O(P)$ time. The reason we can say it's this complexity is that in all loops, every pixel is operated on exactly once as checking other indexes and adding values happens in constant time.

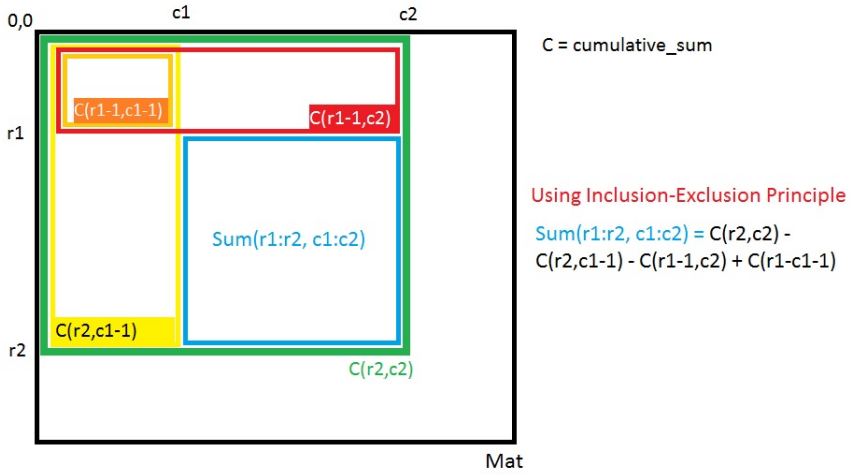

Figure 1: A visualisation of how the final value can be calculated from the cumulative brightness array (stackoverflow.com/questions/39937212/maximum-subarray-of-size-hxw-within-a-2d-matrix)

Part 3 - Darkest Path

Explanation

Computing the darkest path was one of the more interesting parts of this project, and I managed to make it run in linear ($O(P)$) time by treating it like a flood fill test. No graph or priority queues were required.

Correctness

The value of the darkest path is simply the highest pixel brightness that must be encountered when traversing the image from (ur, uc) to (vr, vc). To begin with, the value `darkest` is assumed to be 255 and then tested by attempting a flood fill of all pixels darker than that value. If the destination is found, then we check the next lowest brightness until a value is found such that the destination can not be reached with the restriction. Like in `floodFillCount()`, a set retains which positions have been visited and a stack maintains where to backtrack to if a dead-end is reached. This test runs at most 255 times, but only if the colour appears in the image at least once. The loop skips immediately if the destination is reached, and as soon as a colour is found that can not be avoided, the method returns it.

Complexity

By this treatment, the value of the darkest path can be found in $O(P)$ time. Like the flood fill, this algorithm visits every pixel once however this time the flood fill is operated for every colour, which in this case is a finite number of times (256). Because of this, the run-time for this algorithm increases linearly with input size. As mentioned before, some optimisations were made so that the loop does not run unnecessarily when the answer has been found.

Additionally, before running the flood-fills, a boolean array is created to record which of each colour is in the image so that the flood fill only runs when checking one of these colours. Filling this array also requires $O(P)$ time.

Part 4 - Brightest Pixels in Row Segments

Explanation

For this part a simple naive algorithm is used, that iterates through the queries and checks each against the image to get the maximum brightness.

Correctness

In just two nested loops, each query is accurately parsed and the correct segment checked in the image. This is guaranteed to give the correct output every time.

Complexity

This algorithm is $O(QC)$, as for each query it views it views a number of columns which is at worst C.