CITS2200 Project Report

Arun Muthu

May 26, 2020

Contents

1 FloodFill Count

Correctness

floodFillCount uses a non-recursive depth-first search to search for every contiguous pixel matching the brightness of the original pixel. The inductive step is as follows:

Remove the top-most pixel from the stack and check the pixels above, below, left and right. For any pixel that matches the brightness of the original pixel, convert it to black and push it onto the stack. Increase the count by 1 after this is done.

Since the first original pixel is converted to black and pushed onto the stack, the algorithm will convert its matching neighbours to black, the neighbours' matching neighbours to black and so on. Count is incremented by one for every pixel removed from the stack, and hence once the stack is empty (all suitable pixels processed) the algorithm will return a correct count.

The algorithm does take into account the scenario where the starting pixel is black, in which case 0 is immediately returned.

Time Complexity

The algorithm first creates an empty stack and pushes the original pixel onto it. The algorithm then uses a while loop to perform the depth-first search, stopping when the stack becomes empty. For every iteration of the while loop, the current pixel's 4 neighbours are examined with array accesses and count is incremented by 1. Let these constant time operations be denoted by the variable c. In the worst-case scenario, every pixel in the image matches the brightness of the original pixel and thus the stack will remain non-empty until every pixel in the image is examined. In other words, the while loop will have to execute p times to examine each pixel.

$$\begin{aligned} time &= p \times c \\ &= \mathbf{O(p)} \end{aligned} \tag{1}$$

Hence floodFillCount's worst-case complexity is $\mathbf{O(p)}$.

2 Brightest Square

Correctness

The algorithm starts by finding the sum of every possible subarray of length k in each row and placing the sums in a 2-dimensional integer array *row_sums*.

a1	a2	a3	a4	a5
b1	b2	b3	b4	b5
c1	c2	c3	c4	c5
d1	d2	d3	d4	d5

$\longrightarrow$

a1+a2+a3	a2+a3+a4	a3+a4+a5
b1+b2+b3	b2+b3+b4	b3+b4+b5
c1+c2+c3	c2+c3+c4	c3+c4+c5
d1+d2+d3	d2+d3+d4	d3+d4+d5

It then finds the sum of every possible subcolumn of length k from each column in *row_sums*. The maximum of these sums is the brightness of the brightest square and is the returned answer.

$$max(\sum_{i=0}^{C-1} \sum_{n=0}^{C-k+1} \sum_{j=n}^{k+n-1} row\_sums[j][i])$$

This is because the sum of each subcolumn represents the sum of one k*k square in the image. For example, the sum of the first subcolumn (row 0 to k-1) in the left-most column of row_sums would represent the sum of the top-left most square, the sum of the 2nd subcolumn (row 1 to k) would represent the square one row below, and so on.

The algorithm is correct because it works under the same principle as a brute force solution, i.e. it finds the sum of every possible k*k square and returns the largest sum.

Time Complexity

The algorithm is broken down into 2 main parts, calculating row_sums and calculating the sums of every possible square. The first stage is as follows:

```
for (int i = 0; i < n_rows; i++) {
   int row_sum = 0;
   for (int j = 0; j < k; j++) row_sum += image[i][j];
```

```
    row_sums[i][0] = row_sum;
    for (int j = 1; j < n_cols - k + 1; j++) {
       row_sum += image[i][j + k - 1] - image[i][j-1];
       row_sums[i][j] = row_sum;
    }
 }
```

$$
\begin{aligned}
time_1 &= R(k + (C - k)) \\
&= R * C \\
&= \mathbf{O(p)}
\end{aligned}
\tag{2}
$$

In the 2nd stage:

```
for (int i = 0; i < n_cols - k + 1; i++) {
  int current_sum = 0;
  for (int j = 0; j < k; j++) current_sum += row_sums[j][i];

  max_sum = Math.max(current_sum, max_sum);
  for (int m = 1; m < n_rows - k + 1; m++) {
     current_sum += row_sums[m + k - 1][i]-row_sums[m - 1][i];
     max_sum = Math.max(current_sum, max_sum);
  }
}
```

$$
\begin{aligned}
time_2 &= (C - k + 1)(k + (R - k)) \\
&= (C - k + 1) \times R \\
&= O(RC) \\
&= \mathbf{O(p)}
\end{aligned}
\tag{3}
$$

Both stages are $\mathbf{O(p)}$. $\therefore$ the algorithm's overall complexity is $\mathbf{O(p)}$. The reason for the improved time efficiency compared to a brute force solution is mainly attributed to the way sums of subarrays and subcolumns are calculated. Instead of performing repetitive additions, the earliest element is subtracted from and the latest element is added to the previous sum.

3 Darkest Path

Correctness

The Darkest Path algorithm uses a Priority First Search to find the darkest path. The essence of the search is based on one property. The brightness of a pixel (and thus path) is defined by the brightest pixel it has encountered in the path from the source to itself. This information is stored in the brightness_key array and every update to it is based on one of two outcomes. The pixel's own brightness is the brightest it has encountered thus far or the brightest pixel in the path is. Due to the inequality $max\_brightness \lhd brightness\_key[neighbour]$, the algorithm prioritises **decreasing** the maximum brightness each pixel has encountered.

```
int max_brightness = Math.max(brightness_key[element],
    image[neighbour/n_cols][neighbour%n_cols]);

if (max_brightness < brightness_key[neighbour]) {
    brightness_key[neighbour] = max_brightness;
    pqueue.changePriority(neighbour, max_brightness);
}
```

Hence the brightness_key array always contains an upper bound on the lowest brightness a pixel has encountered. Due to the nature of a priority first search algorithm, pixels are dequeued only when the upper bound has converged to the lowest possible value and thus the algorithm is guaranteed to construct a search tree which emphasises on darkest paths. As a result once (vr, vc) is dequeued from the heap, its value in the brightness_key array will be the brightness of the darkest path from (ur, uc) to (vr, vc).

Time Complexity

Note: The height of a balanced binary tree is log n if there are n elements. The priority first search is broken down into two main operations, dequeueing and updating the brightness_key array. Each dequeue of a pixel from the priority queue is followed by a heapifyDown() operation, which ensures the heap property is still maintained. This is O(log p) in the worst case when the top element is continuously switched with the element below until it reaches the bottom of the heap. For each pixel that is dequeued, its 4 pixel neighbours (left, right, top and bottom in worst case) are examined. If a pixel's brightness_key is updated, this requires a heapifyUp() operation which also restores the heap property. This is O(log p) in the worst-case scenario if an element at the lowest level is brough to the top of the tree. Hence, dequeueing and updating the key array are both O(log p) operations for a single pixel. In the overall worst case scenario, every pixel will be dequeued and every neighbour will have its priority be updated before (vr, vc) is dequeued and the algorithm stops. Therefore the following time complexity is achieved:

$$\begin{aligned} time &= \text{all pixels dequeued} + \text{all neighbours } (4 \times p) \text{ updated} \\ &= p \times logp + 4p \times logp \\ &= 5p \times logp \\ &= \mathbf{O(p\ log\ p)} \end{aligned} \tag{4}$$

4 Brightest Pixels in Row Segments

Correctness

5 Appendix

References

[1] GeeksforGeeks A computer science portal for geeks
`https://www.geeksforgeeks.org/print-maximum-sum-square-sub-matrix-of-given-size/`

[2] Software Engineering Stack Exchange
`https://softwareengineering.stackexchange.com/questions/197016/retrieving-maximum-value-from-a-range-in-unsorted-array`