Programming Project 2020
CITS -Data Structures and Algorithms

Wei Yang (21220208)

Contents

1. Introduction

This individual programming project applies four method to grayscale images which can be seen as a 2D int array. The array is indexed first by row, then by column. Every row in the array will Be the same length.. Every element in the array will be non-negative and no greater than 255. A value of 0 represents a black pixel, and a value of 255 represents white.

- Flood-Fill-Count: Compute the number of pixels that change when performing a black flood-fill from the pixel at (row , col) in the given image. Breadth-first graph search Algorithm(BFS) is used in solving this problem and get a complexity of O(P).

- Brightest-Square: Compute the total brightness of the brightest exactly k*k square that appears in the given image. Two queue are employed for storing data in process and get the data from the queues if needed. This allows each pixel will be examined and handled once during the process, which gives a complexity of O(P).

- Darkest-Path: Compute the maximum brightness that must be encountered when drawing a path from one pixel to another. Dijkstra's algorithm is used to calculate the results. It can be solved in time O(P log P).

- Brightest-Pixels-In-Row-Segment:Compute the results of a list of queries on the given image. An array is used to save segment tree for each image row, and combine those saved segment to the given query. It gives an complexity of O(P+Q log C).

2. Flood Fill Count

2.1 Algorithms

Breadth-first graph search Algorithm (BFS) is what is used to get the answer. The pixels can be seen as vertexes of graph and they are possibly linked with its up/down/left/right pixel as the requirement. However, there are some constrains for this particular method.

- The contiguous pixels must be in the range of the image. It means the index can not be less than 0 or exceed length or width of the image.
- Only the pixel with the same color(value) can be connected.

Then this problem has been converted to a graph searching problem. In the graph, the given pixel and all its contiguous pixels with the same value are connected. Then if another constrain is added:

- One pixel can only be visited once(visited pixels are set to be 1 else set to be 0)

The contiguous region will be traversal with each pixel can be only visited once.

As it is a dark painting, if the selected pixel is 0, it means it will not change so does its contiguous pixels with the same value. 0 is returned in this case. Otherwise, a queue is set up to store the coordinates of pixel with the same value of the given pixel.

1. The coordinates of given pixel is added in the queue as a point(containing x&y) and set the color of it to 1(visited pixel).
2. If the queue is not empty, get the first point out of it and get the value of x&y.
3. Check four possible connected pixel(up/down/left/right), if they meet the 3 constrains mentioned above it means the point is an new pixel that will be painted black(connected&with same color&not been visited&within the range of the image). Then it will be added in the queue. If the there are multiple adjacent pixels meet the requirements, add them in order.
4. Pick a point from the queue if the queue is not empty and do the same thing as step 3, until the queue is empty(no more pixel that meet all constrains has been found).
5. Count total times add in or remove from the queue and return the value. As each contiguous pixels with the same value will be visited , added and removed only once. The times of add or remove is equal to the number of pixels that change when performing a black flood-fill from the given pixel.

2.2 Complexity

For selected pixel and all contiguous pixels of the same colour, each of them is visited with a number of operations. For each pixel steps are list below:

1. Add to the queue (O(1))
2. Change the color to 1(has been visited) (O(1))
3. Remove from the queue (O(1))
4. Count the number (O(1))
5. Check four directions and check if each direction meets the 3 requirements(with int the range,not been visited, with the same color as the given pixel) (O(12))

For the worst case all pixels in the image has the same value which is not 0. Then P pixels will be visited which gives a total complexity of O(16P). As 16 is a constant the complexity is equal to O(P).

3. Brightest Square

3.1 Algorithms

The starting point is to calculate each k*k square in the image and compare them one by one and finally get the biggest result. When R&C>>k, the number of square in the image is

approximately equal to P. So the method gives a complexity of O(k^2P).This is because a pixel is added by nearly a number of k^2 squares if k approaching infinite.

So, if the goal is O(P), each pixel can only be visited once and the number of related operations for one pixel should be a constant. A data structure should be used to store all the useful information for current square and used them for the next square calculation without sum every pixels in squares again and again(most of the elements are overlap).

The method are designed to move square from the top leftmost to rightmost. When the square touch the right boundary, it will be one row downward and back to the leftmost position. Then calculate the sum row by row until the square reach the bottom rightmost corner. In the process two queue(queue1&queue2) are used to store previous results.

1. Before the loop calculation, the value of top leftmost square and the other first row of squares should be calculated.

- Simple sum all the pixels of the top leftmost square, as none of the pixels has been visited. During the calculation, add the sum of each column in two queues separately.
- Then slide the square to right one pixel. The sum of the new square is equal to the sum of previous square plus the sum of new column of length k on the right(add sum of new column in both queue) minus the sum of leftmost column of last square(in the front of queue1). Show as Figure1.
- Repeat the process (calculating sum of new column in and add the sum to both queue,then remove the first element from queue1) until the square reach the rightmost of the first row. New sum of square is equal to last sum+added column-deleted column(elements removed from queue1). Compare the new sum with the previous max. If bigger replace it else keep the previous one.
- Empty queue1 after one row operations.

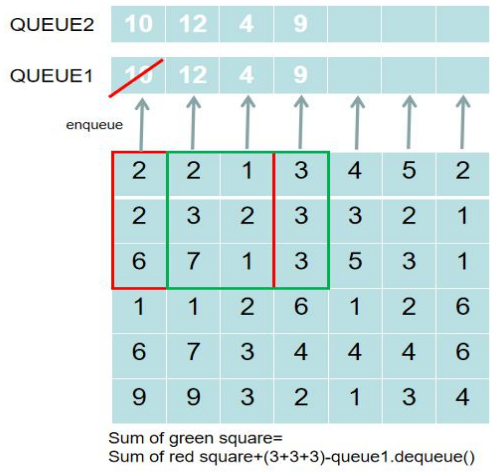

Figure1.Illustration of sliding square

After all these steps, the max of the first row sum is stored in a variable, queue1 and queue 2 are as in Figure2. Queue1 is empty and Queue2 contains the sum of each column for the first

k rows.

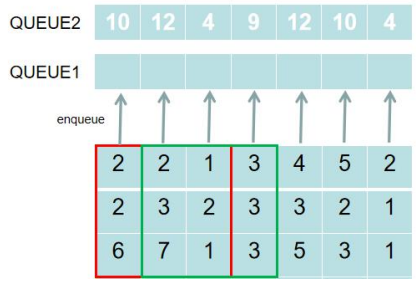

Figure2.Illustration of two queues after first step

2. Start the for loop: Each cycle contains one downward and(C-k) times of leftward move.There are totally (R-k) cycles.

Downward:

- Calculate sum of each of new column by Queue2.dequeue()+downward new pixel - previous top pixel. As is shown in Figure3.
- Add sum of each new column to queue1 and queue2.
- Repeat the process for k times and get the sum of k columns to be the sum of the downward square as the initial for this row and compared it with the max. The status of queue1 and queue2 are shown in Figure3.

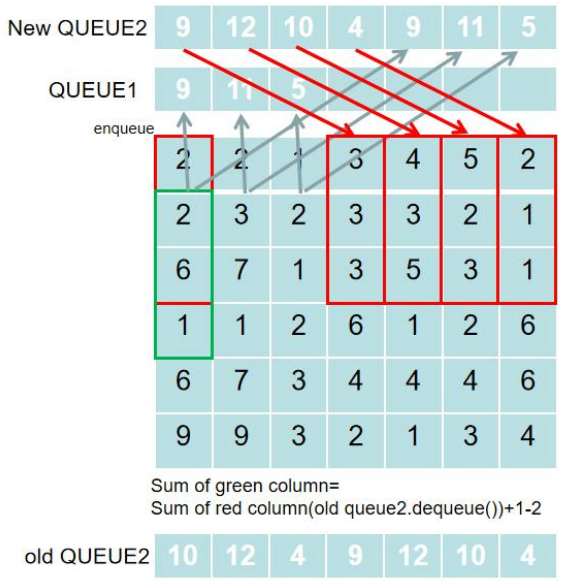

Figure3 Illustration of calculate new column and status of two queues.

Leftward:

- Calculate sum of the added column. As mentioned above in the downward step1, it can be calculated by queue2.dequeue()+downward new pixel - previous top pixel(PS: (9+6-3)for the fourth column).
- Add sum new column in both queues.

- Get the sum of deleted column: Queue1.dequeue();
- Get the sum of the new square = initial for this row+added column-deleted column
- Store the sum of new square to be used as the initial for the next square and compare it with the max(if bigger replace else remain)
- Repeat the process for (C-k)times until it reach the right boundary.
- Finally empty the queue1 and then start the next cycle.

At the end of each cycle, queue1 is empty and queue2 stores all the sum of columns for the previous square row(k rows). Execute the cycle for (R-k) times until finally reach the bottom right corner. All the possible squares are calculated and compared. So the max is now the sum of the Brightest Square.

3.2 Complexity

1. For the first square, it is calculated by columns.For each column:

- Sum the column:O(k)
- Add them in two queue:O(1)
- Make it to be he max: O(1)

For step 1, there are k columns in the square, so the complexity is $O(k^2)$

2. For the first row of square,it is calculated by move.For each move:

- Calculate the sum of added column:O(k)
- Add them in two queue:O(1)
- Get the deleted column by dequeue: O(1)
- Sum the result of new square: O(1)
- Compare it with max: O(1)
- Empty queue1: O(k)

For step 2, there are C-k movements. When C>>k&(C&K) approaching infinite , the complexity is O(Ck)

3. For the rest of calculation, it is calculated by cycles. For each Cycles:

Move downwards:

- Queue2.dequeue():O(1)
- Get the sum of a column:O(1)
- Add them in two queue:O(1)
- Renew value of sum: O(1)

For k times calculation the complexity is O(k)

Compare the downwards Square with max: O(1)

Move leftwards:

- Queue2.dequeue():O(1)
- Get the sum of a column:O(1)
- Add them in two queue:O(1)

- Get the deleted column for dequeue: O(1)
- Renew value of sum: O(1)
- Compare sum with max: O(1)

There are C-k movements. The complexity is O(C)

For step 3, There are totally (R-k) cycles. When R>>k&(R&K) approaching infinite , the complexity is O(CR+Rk+R) which is equal to O(P)

Finally the complexity of all 3 step is O(P+k^2+CK). When R&C>k it is equal to O(P).

4. Darkest Path

4.1 Algorithms

As is mentioned in Flood Fill count, the image can be seen as a graph with each vertex has 4 connected vertex(up/down/left/right) if they are in the range of the image. The weight to the next pixel is the value of that pixel.

In this problem, an algorithm similar to Prim's algorithm and Dijkstra's algorithm are used. An greedy algorithm takes the current best option (the darkest pixel connected to the detected pixels) and then update new options from new pixel. Repeat the process until the best option is the destination then return the biggest value of detected pixels. That will be the answer to the question.

In this algorithm, a heap is used to store all possible options that can be choose at current stage.
All values of pixels connected to current detected ones are added in the heap, the current smallest pixel will be taken out of it. Starting from the original pixel, then the up/down/left/right pixels will be added into the heap,if they are in the range of image and not be added before. Finding the smallest of these added pixels and add its up/down/left/right pixels with two requirements as above. Repeat the process until the destination are taken out of the heap. Then break the loop and return the current biggest detected value.

To prove its correctness, three requirement should be meet and the end of the program.

1. The detected pixels can form at least one path to the destination.
2. The biggest value detected should on the path to the destination.
3. There is no other path with smaller biggest value.

If we can prove they are correct, then the answer is the right one.

The first one is easy to prove. As we can only detect pixel connect to detected pixel(the second detected pixel most be connected to the original one and the third must connect to either the first or the second), when it reach the destination, there must be at least one path to the original.

To prove the second condition, we assume that the biggest value detected is not on the path from origin to the destination. Then, when the path detect the biggest value pixel, there are several options. Since the algorithm will take the smallest in current options, at the time the biggest value pixel are detected, other option must be bigger than it(stage1). At this time, the destination has not been reached.

As I assumed, the biggest value is not on the path to the destination. So if the desalination is to be reached, it has to go back to stage1 and take at least one of the other options. However, other options are bigger than the biggest value. It is contradict to the given condition, either we can not reach the destination or the current biggest value are not the biggest one. So it proves that the biggest value must be on path to the destination.

To prove the third question, I assume there is on path that can reach the destination with the maximum value less than the current biggest value. As the algorithm takes the smallest option, the current biggest pixel should not be reached before the destination is reach and the program finish. However it is reached as given. They are contradict. So there is no way to reach the destination with a smaller maximum passing pixel value.

So the algorithm will give us a correct answer to the question.

4.2 Complexity

For each of the detected pixels including the original, they will go through the following operations:

- Add to the heap: O(log P)
- Set to be detected：O(1)
- Remove from the heap: O(log P)
- Check the four connected pixels: O(1)

For the worst case (the destination pixel has the largest value in the image), the process will be repeated P times. It gives an complexity of O(2P*log P) which is equal to O(P*log P).

5. Brightest Pixels In Row Segments

5.1 Algorithms

There three way to solve the problems

1. Do not store any information and calculate the maximum value by comparing the pixels in

each query segment one by one. That gives a complexity of O(QC), which is suitable for large image with few query.

2. Save the results of every possible enquiry in a 3D array. The method to build the array is shown in Figure4. Then each query will only take O(1). It finally gives a complexity of O(PC+Q), which is suitable for limited image and infinite query.

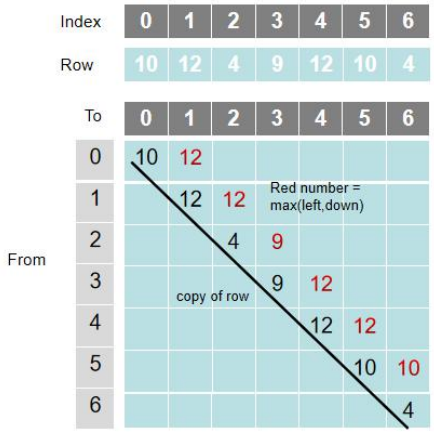

Figure4. Illustration of 2D array construction for a row

3. . Save the maximum of segments of different sizes(from whole row, half of the row, quarter of the row to each pixels). Then for each query , the range of it can be combined by the least use of those segments and get the answer by comparing these combined segments. This algorithm is used in this project and details are explained below.

This algorithm contains two steps:

- Build a segments tree containing every half segments when dividing the row or sub-row by two.
- Use those stored segments to combine the queried segment.

Step one: Building the segment tree

Array is used to store the segments tree, because the number of elements is know to be $2C-1=((C-2^n)*2+2^n+2^{n-1}+2^{n-2}$ +...2+1)(when $2^n<C<2^{n+1}$)and the only options for the array is inputting value and getting value. This two operation take O(1). The value of a row pixels will be input to the last C position of the tree. The rest position will be determined by comparing its right and left child and get the bigger value from the bottom to top. For number of elements is equal to 2^n and not equal to2^n , the trees construction process are shown in figure5.

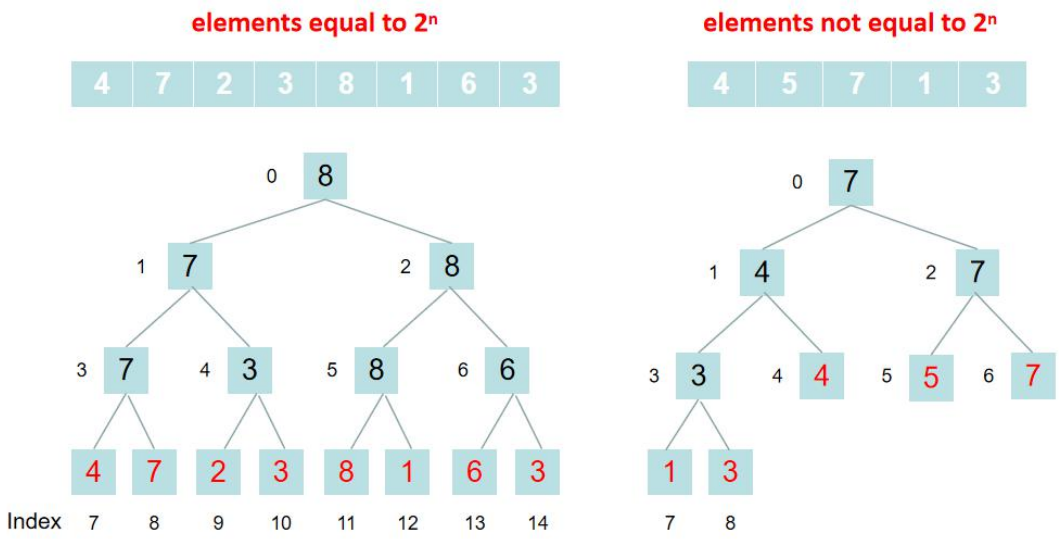

Figure5. Construction of segment tree

Step 2: Combine the queried segment

For the segment tree:

- Each node(except for leave node) must have both left and right child.
- The number of levels is O(log C).
- If the index of a node is odd, this is the left child of its parents and index+1 is the index of the right child and the index divided by 2 is the index of its parent.
- If the index of a node is even, this is the right child of its parents and index-1 is the index of the left child and the index divided by 2 is the index of the right node of its parent or the leftmost node on its own level(if the node is the rightmost one on this level).

Then we can use these features to combine segments into required query.

1. Find the corresponding index for staring and ending pixels(finding the corresponding index for l and u in a query) in segment tree. (l=l+C-1,u=u+C-1)
2. If l=u, there is no segment on this level in range(l,u), then return the current maximum found(0 if l&c are index of leaves, which means no elements in the range of query).
3. If l<u, it means there is at least one segment on this level full in the range. It start from the leaves level (red numbers in Figure5) to the top(their parents).

We combine the query starting from using the segments on lowest level of the tree(value of single pixel) and then level up(double the size of segments) one by one until to the top(the entire row) . When l=u, it means there is no segments on this level fully in the range(l,u) or l&u are refer to one node at this level, then the calculation will stop and the result is got.

For the leaves level (the first cycle):

- If l is odd, it means it is the left child of its parents. The given range will fully contain its parent range if l/2 != u/2. So we can directly find its parents without get any segment at this step. If l/2 = u/2, it means u=l+1 and u will be even. Then we determine the max as

index(u-1) segment. Actually,only one pixel is in the query for this case.

- If l is even, it means it is the right child of its parents. As l is the left point of the segment, the left child of its parent is not in the query range. So the range of its parents are party in the range of query and the inner party is the right child itself. Then we took the value from the tree according to the index and compare it with the current maximum value. When l divided by 2, it will point to the left node on its parents level(next segment may fully contain in the query range).
- If u is odd,it means it is the left child of its parents. As item of index u is not in the query range. We don't do anything for u.
- If u is even,it means it is the right child of its parents. Then we take index(u-1) segment, which is the left child of its parent as a possible maximum value and compare it with the current maximum. Because u is not in the range of query, u-1 will be the only half of its parents in the range.

For upper level:

After first step, l and u will be divided by two.

- If l was odd, after dividing by two it will point to its parents. If l was even, index l segment was solved and after dividing by two it will point to the right node of its parents.
- If u was odd, after dividing by two it will point to its parents. If u was even, index u-1 segment was solved at last step, after (u-1)dividing by two it will point to the its parents as well.

The new l and u are considered to be new leave nodes(we have solve lower level segments and will not visit downward later on for this query) as in step 1. Then repeat the process of step 1 until l=u, which means there is no segment in the tree (segment length of 2^{n-1} elements for the nth cycle) which are full contained in the range of query at current level . Then We have combined the query with segments in our segment tree. Then the maximum number of the combined segments are the final answer.

5.2 Complexity

For constructing the segment tree of an image row:

- 2C-1 elements were added in an arr: O(2C-1)
- Comparing for C-1 times: O(C)

There are R rows in the image which gives a total complexity of O(3RC) which is equal to O(3P) = O(P).

For each query:

The distance(u-l) will be divided by two for each cycle until u-l=1. For the worst case, there will be log C +1 cycles for one query.

For each cycle:

- Make comparison if u or l is even: O(1)
- If u is even u--: O(1)
- Dividing u and l by 2:O(1)

So, for each cycle, the complexity is O(1), and for log C+1 cycles(one query), the complexity is

O(log C). There are totally Q queries, which gives a complexity of O(Q log C).

Finally , the complexity of the algorithm is O(P+Q log C).

6. Conclusion

All the Algorithms used is valid for the specific question and the complexity of each question are listed below:

- Flood-Fill-Count: Breadth-first graph search Algorithm(BFS) is used in solving this problem and get a complexity of O(P).

- Brightest-Square: This allows each pixel will be examined and handled once during the process, which gives a complexity of O(P).

- Darkest-Path: Dijkstra's algorithm is used to calculate the results. It can be solved in time O(P log P).

- Brightest-Pixels-In-Row-Segment: An array is used to save segment tree for each image row, and combine those saved segment to the given query. It gives an complexity of O(P+Q log C).

7. Reference

Priority queue in java
https://www.geeksforgeeks.org/priority-queue-class-in-java-2/

Problem Solving in Java: Sliding Window Algorithm-Chhaian Pin, 2019
https://medium.com/@chyanpin/problem-solving-in-java-sliding-window-algorithm-f333d362478b

Segment Tree,2018
http://www.baidu.com/link?url=b2iRO-bTLNyjGDslzeya6IU9ZOi_eINn188WcbzL5Up4foffGUASjD9lvESRtD04&wd=&eqid=ecf28c710004fe44000000065ecfa748