Programming Project

COT 5405 - Analysis of Algorithms Spring 2018

Cheng Li 31508398

Task1 :

This task is similar to the LCS problem in dynamic programming. Let H[i,j] records the height of (i, j). In order to reduce the space complexity, I did not create another matrix to store the edge length of the maximum square with the right corner of (i,j). Instead, I just overwrite the value in H[i,j] in the code. However, to avoid confusion, I`ll use the notation A[i, j] to specify the edge.

$$H[i,j] = \begin{cases} 0 & \text{if } H[i,j] \text{ !=C} \\ 1 & \text{if } i = 1 \text{ or } j = 1 \\ \min(H[i,j-1], H[i-1,j], H[i-1,j-1]) + 1 & \text{otherwise} \end{cases}$$

The correctness of this optimal structure can be proved with the following relations:

If H(i,j) < C, then the edge of current square cannot expand anymore and we count the edge of a new square from (i,j). For the first row/column, the edge is fixed. Let k = min (A[i-1, j], A[i, j-1], A[i-1,j-1]). Then the following relations hold:

$$H[i-k...i-1, j-k...j-1] \subset H[i-k...i, j-k...j]$$
$$H[i-k...i-1, j-k+1...j] \subset H[i-k...i, j-k...j]$$
$$H[i-k+1...i, j-k...j-1] \subset H[i-k...i, j-k...j]$$

The time complexity is O(n^2).

The space complexity is O(n) because we overwrite on the H[i,j] which is input.

Task2.

In this task, I use an array named wid[] to record the width of the candidate rectangle. The width is decided by how far it can go up, assuming the outer loop is i, which means it scans row by row and the wid[] is updated in each row. The following formulation shows how it works:

$$Wid[j] = \begin{cases} 0 & \text{if } H[i,j] \text{ !=C} \\ wid[j] + 1 & \text{if } H[i,j] = C \end{cases}$$

Then the main problem can be transformed into the subproblem that calculates the largest rectangle in each row. To make it simple, it can be regarded as Area[i-1, j] ⊂ Area[i, j] because Area[i, j] = wid[j]*length. So, the strategy to find out the largest rectangle is using two loops. For each j, use index k=j+1 to find out the number of consecutive wid[j] satisfies two conditions: wid[k] != 0 and wid[k] > wid[j]. Then the number is the length of the rectangle. So, we can calculate the area by multiplying wid[k]*length. Each time we get a larger area, we replace the previous area.
Since we need two loops for each row and one loop for columns, the complexity will be Θ(NM^2). The space complexity is O(M) which is taken by the wid[].

Task3.
The recursive formulation is same as Task2 but the data structure is changed.
In order to get the reduced time complexity, I need to eliminate one loop. The general idea is same as Task2 but use a stack instead of a simple array. The strategy is :

1. push the index with the width larger than the previous one.
2. pop all the indexes with width smaller than the current index. Then the difference between the peek index in the stack and the current index becomes length of the candidate rectangle.

Since one loop is eliminated, the time complexity becomes Θ(NM). Space complexity is still O(M).

In Task4, I used similar algorithm as before , but do i,j loop once to get the length of candidate rectangle for each point and then do j,i loop to get the width of candidate rectangles for each position. Then I use another i,j loop to get the area for each position but ignore the violence in cross directions, like i-1,j-1 and i, j. This procedure above has the similar recursive formulation as before: len[i, j-1] ⊆ len[i,j] and wid[i-1, j] ⊆ wid[i,j]. Then, in order to eliminate those rectangles have violation of C in cross directions, I use two loops which have the range of len[i,j] and wid[i,j].

```
for(int i = 0; i < N; i++) {
    for (int j = 0; j< M; j++ ) {
        int hi = area[i][j]/len[i][j];
        int maxhi = Integer.MIN_VALUE;
        int minhi = Integer.MAX_VALUE;
        int k=i;
        int l=j;
        for(; k < i + hi; k++) {
            for (l=j; l < j + len[i][j]; l++ ) {//k=i , l is not
                if(height[k][l] > maxhi) maxhi = height[k][l];
                if(height[k][l] < minhi) minhi = height[k][l];
        }
    }
```

I pick the min height value and maximum height value in that area and compare it to C. I only record those areas composed of k and l where the max height - min height < C.

The time complexity is Θ(NM^2C) since I have another loop with range C inside the i, j k loop.

```
for(int m = 0; m < C+1; m++) {
    if (dif[m] > max) {//larger than , not >=
        max=dif[m];
        result[0] = k-1 + 1;
        result[1] = l-1 + 1;
        result[2] = result[0] - hi + 1 ;
        result[3] = result[1] - len[i][j] + 1 ;
    }
}
```

The space complexity is O(CM) considering the array which stores the maximum areas with different c < C.