Previous Next

562. Longest Line of Consecutive One in Matrix

April 22, 2017 | 14.7K views

Average Rating: 4.15 (13 votes)

Given a 01 matrix **M**, find the longest line of consecutive one in the matrix. The line could be horizontal, vertical, diagonal or anti-diagonal.

Example:

```
Input:
[[0,1,1,0],
 [0,1,1,0],
 [0,0,0,1]]
Output: 3
```

Hint: The number of elements in the given matrix will not exceed 10,000.

Solution

Approach 1: Brute Force

Algorithm

The brute force approach is really simple. We directly traverse along every valid line in the given matrix: i.e. Horizontal, Vertical, Diagonal aline above and below the middle diagonal, Anti-diagonal line above and below the middle anti-diagonal. Each time during the traversal, we keep on incrementing the $count$ if we encounter continuous 1's. We reset the $count$ for any discontinuity encountered. While doing this, we also keep a track of the maximum $count$ found so far.

Java Copy

```java
class Solution {
  public int longestLine(int[][] M) {
    if (M.length == 0) return 0;
    int ones = 0;
    // horizontal
    for (int i = 0; i < M.length; i++) {
      int count = 0;
      for (int j = 0; j < M[0].length; j++) {
        if (M[i][j] == 1) {
          count++;
          ones = Math.max(ones, count);
        } else count = 0;
      }
    }
    // vertical
    for (int i = 0; i < M[0].length; i++) {
      int count = 0;
      for (int j = 0; j < M.length; j++) {
        if (M[j][i] == 1) {
          count++;
          ones = Math.max(ones, count);
        } else count = 0;
      }
    }
    // upper diagonal
    for (int i = 0; i < M[0].length || i < M.length; i++) {
      int count = 0;
```

Complexity Analysis

- Time complexity : $O(n^2)$. We traverse along the entire matrix 4 times.
- Space complexity : $O(1)$. Constant space is used.

Approach 2: Using 3D Dynamic Programming

Algorithm

Instead of traversing over the same matrix multiple times, we can keep a track of the 1' along all the lines possible while traversing the matrix once only. In order to do so, we make use of a $4mn$ sized dp array. Here, $dp[0]$, $dp[1]$, $dp[2]$,$dp[3]$ are used to store the maximum number of continuous 1's found so far along the Horizontal, Vertical, Diagonal and Anti-diagonal lines respectively. e.g. $dp[i][j][0]$ is used to store the number of continuous 1's found so far(till we reach the element $M[i][j]$), along the horizontal lines only.

Thus, we traverse the matrix M in a row-wise fashion only but, keep updating the entries for every dp appropriately.

The following image shows the filled dp values for this matrix:

```
0 1 1 0

0 1 1 0

0 0 1 1
```

dp[0]

0	1	2	0
0	1	2	0
0	0	1	2

dp[1]

0	1	1	0
0	2	2	0
0	0	3	0

dp[2]

0	1	1	0
0	1	2	0
0	0	2	3

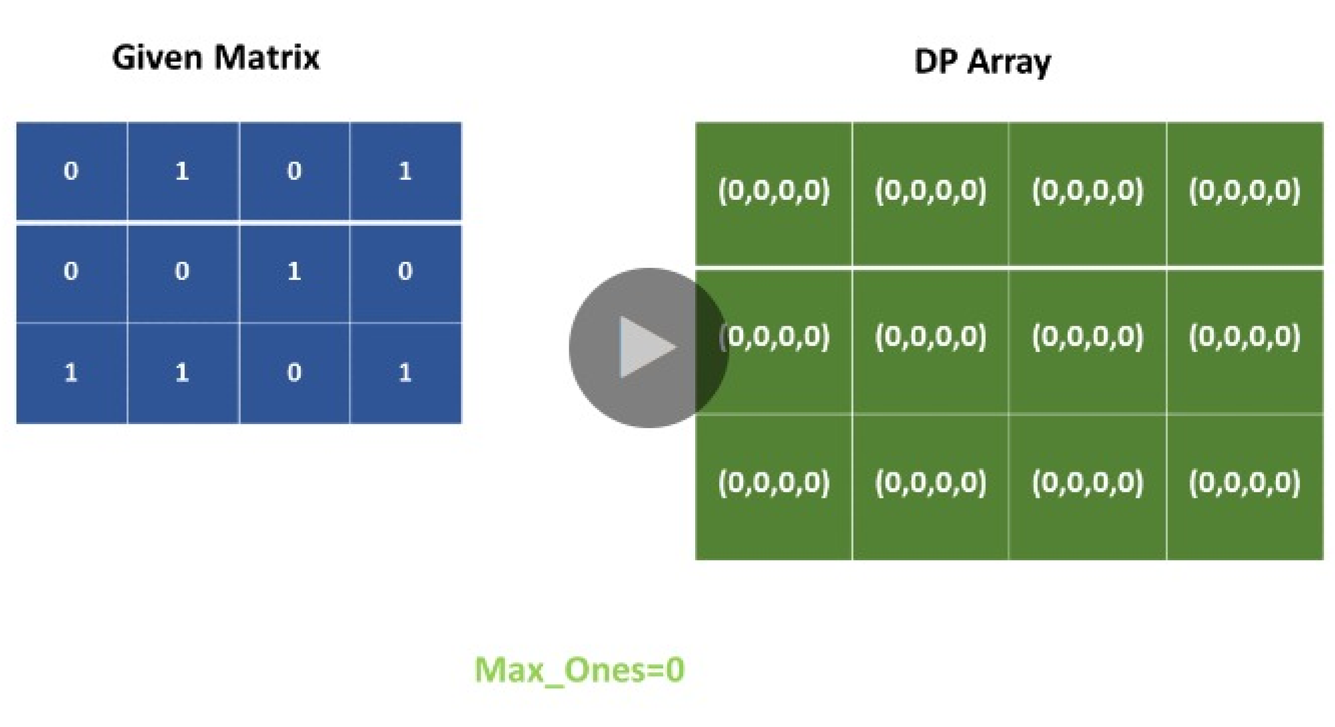
dp[3]

0	1	1	0
0	2	1	0
0	0	1	1

max_ones = max(dp) = 3

While filling up the dp, we can keep a track of the length of the longest consecutive line of 1's.

Watch this animation for complete process:



Java Copy

```java
class Solution {
  public int longestLine(int[][] M) {
    if (M.length == 0) return 0;
    int ones = 0;
    int[][][] dp = new int[M.length][M[0].length][4];
    for (int i = 0; i < M.length; i++) {
      for (int j = 0; j < M[0].length; j++) {
        if (M[i][j] == 1) {
          dp[i][j][0] = j > 0 ? dp[i][j - 1][0] + 1 : 1;
          dp[i][j][1] = i > 0 ? dp[i - 1][j][1] + 1 : 1;
          dp[i][j][2] = (i > 0 && j > 0) ? dp[i - 1][j - 1][2] + 1 : 1;
          dp[i][j][3] = (i > 0 && j < M[0].length - 1) ? dp[i - 1][j + 1][3] + 1 : 1;
          ones =
              Math.max(
                  ones,
                  Math.max(Math.max(dp[i][j][0], dp[i][j][1]), Math.max(dp[i][j][2], dp[i][j][3])));
        }
      }
    }
    return ones;
  }
}
```

Complexity Analysis

- Time complexity : $O(mn)$. We traverse the entire matrix once only.
- Space complexity : $O(mn)$. dp array of size $4mn$ is used, where m and n are the number of rows ans coloumns of the matrix.

Approach 3: Using 2D Dynamic Programming

Algorithm

In the previous approach, we can observe that the current dp entry is dependent only on the entries of the just previous corresponding dp row. Thus, instead of maintaining a 2-D dp matrix for each kind of line of 1's possible, we can use a 1-d array for each one of them, and update the corresponding entries in the same row during each row's traversal. Taking this into account, the previous 3-D dp matrix shrinks to a 2-D dp matrix now. The rest of the procedure remains same as the previous approach.

Java Copy

```java
class Solution {
  public int longestLine(int[][] M) {
    if (M.length == 0) return 0;
    int ones = 0;
    int[][] dp = new int[M[0].length][4];
    for (int i = 0; i < M.length; i++) {
      int old = 0;
      for (int j = 0; j < M[0].length; j++) {
        if (M[i][j] == 1) {
          dp[j][0] = j > 0 ? dp[j - 1][0] + 1 : 1;
          dp[j][1] = i > 0 ? dp[j][1] + 1 : 1;
          int prev = dp[j][2];
          dp[j][2] = (i > 0 && j > 0) ? old + 1 : 1;
          old = prev;
          dp[j][3] = (i > 0 && j < M[0].length - 1) ? dp[j + 1][3] + 1 : 1;
          ones =
              Math.max(ones, Math.max(Math.max(dp[j][0], dp[j][1]), Math.max(dp[j][2], dp[j][3])));
        } else {
          old = dp[j][2];
          dp[j][0] = dp[j][1] = dp[j][2] = dp[j][3] = 0;
        }
      }
    }
    return ones;
  }
}
```

Complexity Analysis

- Time complexity : $O(mn)$. The entire matrix is traversed once only.
- Space complexity : $O(n)$. dp array of size $4n$ is used, where n is the number of columns of the matrix.

Analysis written by: @vinod23

Rate this article:

Previous Next

Comments: 14

Sort By

Type comment here... (Markdown is supported)

Preview Post

ashketchum ★41 August 3, 2018 7:43 AM

The Brute Force is giving better complexity than you Dynamic Programming approaches. Why should I solve the Dynamic programming approach over the brute force (technically)?

12 Share Reply

SHOW 5 REPLIES

melodyincognito ★44 July 5, 2019 11:12 AM

Here is O(nm) solution by just using hash tables:

```
import collections
class Solution:
```

Read More

8 Share Reply

sean46 ★91 April 26, 2017 2:23 AM

Could you add a comment on how the old/prev work to handle the diagonal case?

3 Share Reply

Pengwu550 ★63 February 12, 2019 11:58 PM

the explaination for the third one is poor to understand

2 Share Reply

SHOW 2 REPLIES

Ulfbert ★1 March 30, 2020 5:12 PM

Shouldn't approach 2 the same as approach 1 in term of time complexity?
You are going over the entire matrix once but at each element you are doing 4 times the amount of work compared to when you are going over an element in approach 1.

1 Share Reply

peaceCoder ★452 October 18, 2018 4:24 AM

It took me some while to understand the old and prev variables. Excellent optimized dp solution.

1 Share Reply

vinod23 ★425 April 26, 2017 5:25 AM

@sean46 old/prev is used to store the previous dp[j][2] value. In jth iteration dp[j][2] will be updated, therefore we are storing the value of dp[j][2] in "old" variable so that we can use it in (j+1)th iteration.

1 Share Reply

laosunhust ★6 April 25, 2017 2:29 AM

There is a O(mn) time O(1) space solution as well. check this out
https://discuss.leetcode.com/topic/87416/o-mn-time-no-extra-space-solution

1 Share Reply

SHOW 1 REPLY

rock1270 ★5 October 1, 2018 11:08 PM

Very easy constant space solution with explanation:
https://leetcode.com/problems/longest-line-of-consecutive-one-in-matrix/discuss/176478/O(mn)-with-O(1)-space-with-detailed-explanation

0 Share Reply

y5yeyey ★8 October 12, 2017 1:36 AM

There is another O(mn) time O(1) space solution. Must see!
https://discuss.leetcode.com/topic/106745/java-o-1-space-and-o-mn-time-69-running-time-percentile

0 Share Reply

1 2