329. Longest Increasing Path in a Matrix

610 13

Description (/problems/longest-increasing-path-in-a-matrix/description/) Hints (/problems/longest-increasing-path-in-a-matrix/hints/) Subm

Quick Navigation View in Article (/articles/longest-increasing-path-matrix/)

Notes

Summary

This article is for advanced readers. It introduces the following ideas: Depth First Search (DFS), Memoization, Dynamic programming, Topological Sorting. It explains the relation between dynamic programming and topological sorting.

Solution

Approach #1 (Naive DFS) [Time Limit Exceeded]

Intuition

DFS can find the longest increasing path starting from any cell. We can do this for all the cells.

Algorithm

Each cell can be seen as a vertex in a graph G. If two adjacent cells have value $a < b$, i.e. increasing then we have a directed edge (a, b). The problem then becomes:

> Search the longest path in the directed graph G.

Naively, we can use DFS or BFS to visit all the cells connected starting from a root. We update the maximum length of the path during search and find the answer when it finished.

Usually, in DFS or BFS, we can employ a set `visited` to prevent the cells from duplicate visits. We will introduce a better algorithm based on this in the next section.

Java Copy

Complexity Analysis

- Time complexity : $O(2^{m+n})$. The search is repeated for each valid increasing path. In the worst case we can have $O(2^{m+n})$ calls. For example:

Java Copy

- Space complexity : $O(mn)$. For each DFS we need $O(h)$ space used by the system stack, where h is the maximum depth of the recursion. In the worst case, $O(h) = O(mn)$.

Approach #2 (DFS + Memoization) [Accepted]

Intuition

Cache the results for the recursion so that any subproblem will be calculated only once.

Algorithm

From previous analysis, we know that there are many duplicate calculations in the naive approach.

One optimization is that we can use a set to prevent the repeat visit in one DFS search. This optimization will reduce the time complexity for each DFS to $O(mn)$ and the total algorithm to $O(m^2n^2)$.

Here, we will introduce more powerful optimization, Memoization.

> In computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

In our problem, we recursively call `dfs(x, y)` for many times. But if we already know all the results for the four adjacent cells, we only need constant time. During our search if the result for a cell is not calculated, we calculate and cache it; otherwise, we get it from the cache directly.

Complexity Analysis

- Time complexity : $O(mn)$. Each vertex/cell will be calculated once and only once, and each edge will be visited once and only once. The total time complexity is then $O(V+E)$. V is the total number of vertices and E is the total number of edges. In our problem, $O(V) = O(mn)$, $O(E) = O(4V) = O(mn)$.
- Space complexity : $O(mn)$. The cache dominates the space complexity.

Approach #3 (Peeling Onion) [Accepted]

Intuition

The result of each cell only related to the result of its neighbors. Can we use dynamic programming?

Algorithm

If we define the longest increasing path starting from cell (i, j) as a function

$$f(i,j)$$

then we have the following transition function

$$f(i,j) = max\{f(x,y) | (x,y) \text{ is a neighbor of} (i,j) \text{ and } \text{matrix}[x][y] > \text{matrix}[i][j]\} + 1$$

This formula is the same as used in the previous approaches. With such transition function, one may think that it is possible to use dynamic programming to deduce all the results without employing DFS!

That is right with one thing missing: we don't have the dependency list.

For dynamic programming to work, if problem B depends on the result of problem A, then we must make sure that problem A is calculated before problem B. Such order is natural and obvious for many problems. For example the famous Fibonacci sequence:

$$F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$$

The subproblem $F(n)$ depends on its two predecessors. Therefore, the natural order from 0 to n is the correct order. The dependent is always behind the dependee.

The terminology of such dependency order is "Topological order" or "Topological sorting":

> Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (u, v), vertex u comes before v in the ordering.

In our problem, the topological order is not natural. Without the value in the matrix, we couldn't know the dependency relation of any two neighbors A and B. We have to perform the topological sort explicitly as a preprocess. After that, we can solve the problem dynamically using our transition function following the stored topological order.

There are several ways to perform the topological sorting. Here we employ one of them called "Peeling Onion".

The idea is that in a DAG, we will have some vertex who doesn't depend on others which we call "leaves". We put these leaves in a list (their internal ordering does matter), and then we remove them from the DAG. After the removal, there will be new leaves. We do the same repeatedly as if we are peeling an onion layer by layer. In the end, the list will have a valid topological ordering of our vertices.

In out problem, since we want the longest path in the DAG, which equals to the total number of layers of the "onion". Thus, we can count the number of layers during "peeling" and return the counts in the end without invoking dynamic programming.

Complexity Analysis

- Time complexity : $O(mn)$. The the topological sort is $O(V + E) = O(mn)$. Here, V is the total number of vertices and E is the total number of edges. In our problem, $O(V) = O(mn)$, $O(E) = O(4V) = O(mn)$.
- Space complexity : $O(mn)$. We need to store the out degrees and each level of leaves.

Remarks

- Memoization: for a problem with massive duplicate calls, cache the results.
- Dynamic programming requires the subproblem solved in topological order. In many problems, it coincides the natural order. For those who doesn't, one need perform topological sorting first. Therefore, for those problems with complex topology (like this one), search with memorization is usually an easier and better choice.

Comments: 8

Sort By

Type comment here... (Markdown is supported)

Preview Post

(/haotianliu1801)

haotianliu1801 (/haotianliu1801) ★ 1 September 22, 2018 10:36 PM

Why do we have to increment ans by 1 at the end of DFS in solution #1?

0 Share Reply

(/bip)

bip (/bip) ★ 6 September 5, 2018 7:44 AM

For approach #1 shouldn't the space complexity be O(m+n)?

0 Share Reply

(/ghostfacechillah)

ghostfacechillah (/ghostfacechillah) ★ 26 April 11, 2018 12:54 AM

"Usually, in DFS or BFS, we can employ a set visited to prevent the cells from duplicate visits. We will introduce a better algorithm based on this in the next section." but I did not find which part in the article explains why we don't have to maintain such a visited set. My guess is because the path is increasing, we will never visit a node with smaller value.

1 Share Reply

SHOW 1 REPLY

(/jiaguowmu)

jiaguowmu (/jiaguowmu) ★ 4 March 15, 2018 1:45 AM

Who can help explain why the Time complexity of Approach 1 is O(2^(m+n))?
Thanks.

4 Share Reply

SHOW 2 REPLIES

(/hanyuxin)

HanYuxin (/hanyuxin) ★ 1 December 2, 2017 12:39 AM

Why you add zero as boundaries in the Peeling Onion method, I think it is not necessary

0 Share Reply

(/dexterhu)

dexterhu (/dexterhu) ★ 0 August 18, 2017 3:15 AM

```
class Solution {
public int longestIncreasingPath(int[][] matrix) {
```

Read More

0 Share Reply