Matrix Path Optimizer

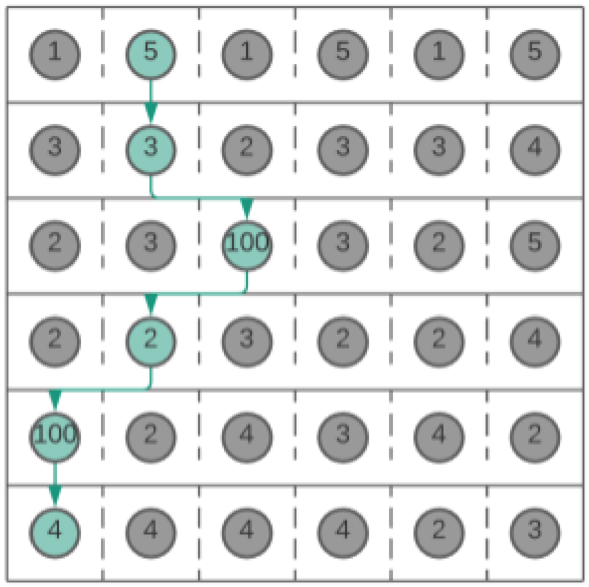
Dynamic Programming Solution for an Optimal 2D Subsequence

By Mason Heaman

1. Introduction

Suppose you are given an M*N matrix representing the aerial view of a truffle farm, where each element represents one square meter. At each row of the field, the farmer may choose to travel one step diagonally to the left, straight down, or diagonally to the right. In order to obtain a maximal yield while harvesting the truffles, an optimal path must be calculated using these conditions. (see Figure 1.1)

Figure 1.1



Example of an optimal path for a matrix

2. Plausibility, Efficiency, and Resources

This algorithm will be made possible using a dynamic programming "bottom-up" approach. This will allow the algorithm to perform at an efficient *O*(M * N) time. Given the following direction and pseudo-code, a junior level engineer should be able to implement the algorithm in 1-2 working sessions at a single workstation.

3. Implementation

3.1 Input

mat- A two-dimensional array holding the number of truffles at a given square mile.

3.2 Output

Positions - the indices at which one should harvest to obtain the maximum number of truffles
Values - the number of doubloons at the positions, respectively
Maximum- the maximum number of truffles that can be picked on a single path.

3.3 Method

Implementation can be separated into two processes: creating a maximum value (*MV)* matrix and finding the subsequence positions/values.

3.3.1 Creating the *MV* Matrix

MV is an auxiliary matrix that provides the largest number of truffles that could be harvested by starting at any given square meter. The optimal harvesting path can then be seen as the maximum value in the first row of *MV*, as this will provide the optimal path starting at the top of the field. To find *MV[i][j]* note that it will be the current value plus the maximum path starting at either *MV[i+1][j-1]*, *MV[i+1][j]*, or *MV[i+1][j+1]*. Array bounds must be accounted for to ensure that we are not trying to access beyond, i.e. accessing the right diagonal of the rightmost element. See the following pseudo-code for reference:

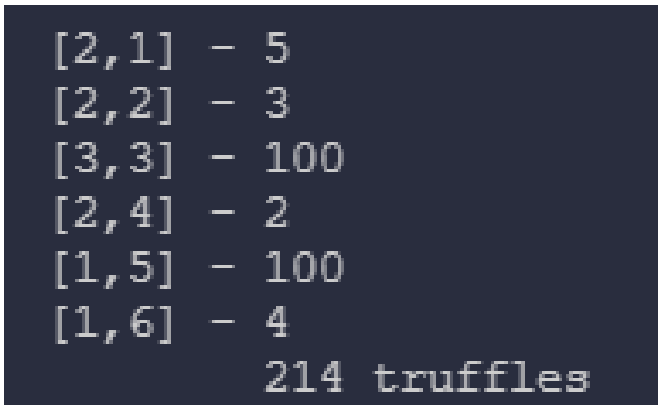
```
GET-MV(mat, minDistance, M, N)
let choices[] be a new array holding the possible moves
 for i=M-2 down to -1:
   for j = 0 to N:
     add mat[i+1][j] to choices
     if leftmost element:
       add mat[i+1][j+1] to choices
     if rightmost element:
       add mat[i+1][j-1] to choices
     else:
       add mat[i+1][j-1] to choices
       add mat[i+1][j+1] to choices
   mat[i][j] += max(choices)
  return mat
```

3.3.2 Retrieving Subsequence Positions & Values

Finding the positions and values that create the optimal path will be done utilizing both *mat* and *MV*. The relationship between the two allows for positions and values to be found as follows:

```
   GET-PATH(mat, MV)
   1. current_index = indexOf(max(MV[0]))//starting index
   2. //print index of starting element and value at mat[index]
   3. for i=0 to arr.size:
   4.   target = MV[i][current_index] - mat[i][current_index]
   5.   current_index = MV[i+1][indexOf(target)]
   6.   //print the index of the current element and value in
mat
   7. print max(MV[0) //maximum path total
```

The expected output of GET-PATH on the matrix in Figure 1.1 is as follows:



```
[2,1] - 5
[2,2] - 3
[3,3] - 100
[2,4] - 2
[1,5] - 100
[1,6] - 4
        214 truffles
```