Doubloon Path Optimizer

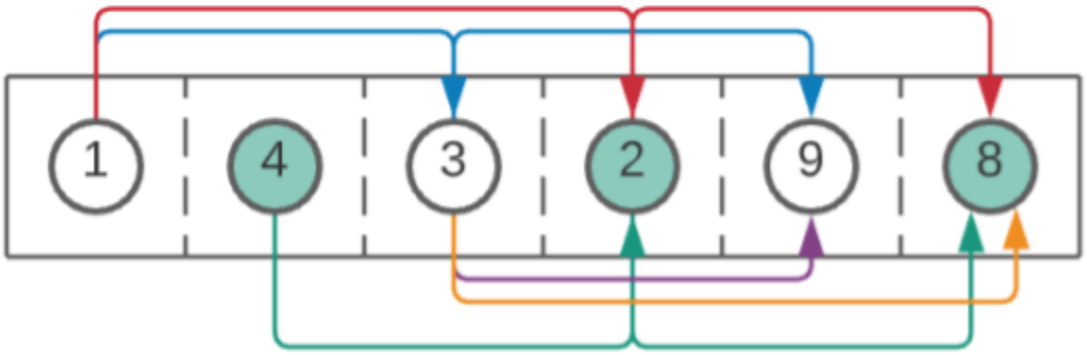
Dynamic Programming Solution for an Optimal Subsequence

By Mason Heaman

1. Introduction

Suppose you are given a path with a length of *M* miles going east to west. This path will have a stash of gold coins at each mile, although they can also be empty. While traveling this path, you can pick up the coins at a stash, but you must then travel at least *MinDistance* before you can pick up another stash. For example, with a *MinDistance* of 2, if you pick up a stash of doubloons at mile 2 you cannot pick up another stash until you reach mile 4. As shown in **Figure 1.1**, there are multiple subsequences of picks that one could choose to make (note that this figure is not exhaustive). We want to end the path with the most doubloons possible. Therefore, our goal is to find the optimal subsequence of doubloons to pick. For example, in Figure 1 the optimal subsequence is the green series at positions 1, 3, and 5.

Figure 1.1



Some of the possible subsequences for the path 1, 4, 3, 2, 9, 8 with *minDistance=2*

2. Implementation

2.1 Input

minDistance - An integer denoting the minimum travel between doubloon picks
arr - An integer array *arr* of length *M*. The value at each index is the number of doubloons at that index. Therefore, an empty stash is denoted with the value 0.

2.2 Output

Positions - the indices at which one should pick to obtain the maximum number of doubloons
Values - the number of doubloons at the positions, respectively
Total - the maximum number of doubloons that can be picked

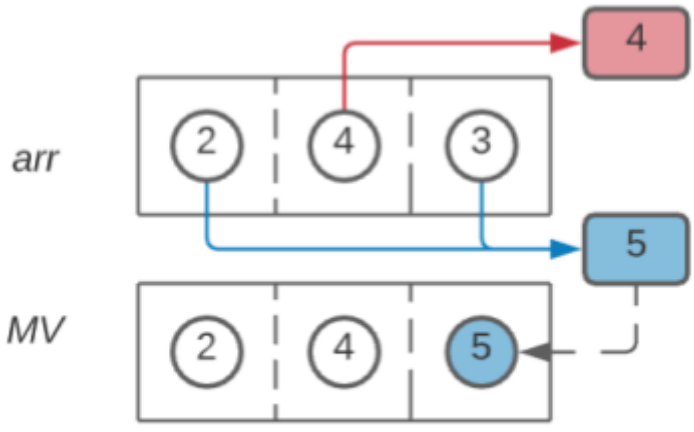
2.3 Method

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

2.3.1 Creating the *MV* Array

MV is an auxiliary array that provides the largest number of doubloons that one could have at any given mile on the path. Therefore, *MV*[*M-1*] will hold the maximum number of doubloons that can be picked through the whole path. To find *MV[i]* note that there two possible values one can have at any given index: the maximum value at mile *i*-1, or the maximum value at mile *i-minDistance* plus the value at mile *i*. We want the maximum of the two; see **Figure 2.1.**

Figure 2.1



Picking at index 2 (*MV*[*i-minDistance*] + *MV*[*i*]) is greater than the max value at the last mile (*MV*[*i*-1*]*) so it gets added to *MV*

This rule allows for the simple and efficient creation of the *MV* array. Use the following pseudo-code as reference:

```
GET-MV(arr, minDistance, M)
1. let MV[0...M+2] be a new array
2. For i=0 to minDistance:
3.    MV[i] = 0
4. for i=0 to n
5.    MV[i] = max(MV[i-1], MV[i-minDistance] + arr[i])
6. return MV[n]
```

2.3.2 Retrieving Subsequence Positions & Values

To find the indices that make up the optimal subsequence we will use the backtracking method. To find the correct indices, step through the *MV* array starting at the last element and handle one of two conditions as follows. If the next value is the same as the current, move on to the next value. If the next value is not the same as the current value, store the index of *arr* that corresponds to the current index *arr*[*i-minDistance*] (this is to account for the zeros padding the *MV* array) and move down to the element at *i-minDistance*. Use the following pseudo-code as reference:

```
GET-POSITIONS(arr, minDistance, M, MV)
1. let inds be a new ArrayList
2. i = MV.length-1
3. while i > 0
4.    if MV[i] == MV[i-1]
5.       i--
6.       continue
7.    else
8.       inds.add(i-minDistance)
9.       i-= minDistance
10.   return inds
```

The *inds* ArrayList will now hold the indices where picks were made for the optimal subsequence. The values for these indices can be easily found as arr[ind[i]] for every index in inds.