Dynamic Programming Solution to the Check Board Problem

Cheng Li, Virgil Pavlu, Javed Aslam

Check Board Problem

We are given a check board with m rows and n columns. There is a cost (penalty) $P[i, j]$ associated with square (i, j). We start at any square in the first row (row 1), and we want to get to anywhere in row m with least total penalty, assuming we could only move diagonally left forward, diagonally right forward, or straight forward.

	1	2	·	·	n
m	1	22	12	14	28
·	34	5	12	23	9
·	8	7	16	25	36
2	11	5	24	3	41
1	10	13	47	96	8

Methodology

(1) Characterize the Structure of an Optimal Solution. The Check Board problem exhibits optimal substructure in the following manner. Consider any optimal path from row 1 to row m. Now consider breaking that path into two subpaths at any square (i, j) on this path.

Claim 1 *The subpath from row 1 to square (i, j) must be an optimal way to move from row 1 to square (i, j), and the subpath from square (i, j) to row m must be an optimal way to move from square (i, j) to row m.*

Proof: By contradiction, suppose that there was a better subpath from row 1 to square (i, j) than the subpath from row 1 to square (i, j) in the optimal solution. Then the the subpath from row 1 to square (i, j) in the optimal solution could be replaced with this better solution, yielding a valid solution to moving from row 1 to row m with smaller costs than the solution being considered. But this contradicts the supposed optimality of the given solution, $\rightarrow\leftarrow$. An identical argument applies to the subpath from square (i, j) to row m in the solution. $\square$

Thus, the optimal solution to the check board problem is composed of optimal solutions to smaller subproblems.

(2) Recursively Define the Value of the Optimal Solution. First, we define in English the quantity we shall later define recursively. Let $C[i, j]$ be the cheapest cost to get from row 1 to square (i, j). In the optimal solution to getting to square (i, j), there must exist some last step, $(k, l) \rightarrow (i, j)$, where (k, l) can only be $(i-1, j-1)$, $(i-1, j)$, or $(i-1, j+1)$. Furthermore, the subpath from row 1 to square (k, l) must be an optimal solution to moving from row 1 to square (l,k), since check board exhibits optimal substructure as proven above. Thus, if $(k,l) \rightarrow (i,j)$ is the last move in the optimal solution to move from row 1 to square (i,j), then $C[i,j] = P[i,j] + C[k,l]$. We don't know which square (k,l) is; however, we may check all 3 such possibilities (square $(i-1,j-1)$, $(i-1,j)$, and $(i-1,j+1)$), and the value of the optimal solution must correspond to the minimum value of $P[i,j] + C[k,l]$, by definition. Furthermore, when $i = 1$, the cost is clearly $P[i,j]$. We thus have the following recurrence.

Claim 2 $C[i,j] = \begin{cases} P[i,j] & \textit{if } i = 1 \\ P[i,j] + \min\{C[i-1,j-1], C[i-1,j], C[i-1,j+1]\} & \textit{other} \end{cases}$

Proof: The correctness of this recursive definition is embodied in the paragraph which precedes it. □

(3) Compute the Value of the Optimal Solution Bottom-up. Consider the following piece of pseudocode, where P is the cost matrix, m is the number of rows, and n is the number of columns.

```
Check(P[ ], m, n)
1  for j ← 1 to n
2      C[1, j] ← P[1, j]
3  for i ← 1 to m
4      for j ← 1 to n
5          C[i, j] ← P[i, j] + min{C[i−1, j−1], C[i−1, j], C[i−1, j+1]}
6          S[i, j] ← arg min_j{C[i−1, j−1], C[i−1, j], C[i−1, j+1]}
7  return C and S
```

Claim 3 *When the above procedure terminates, for all $1 \leq i \leq m$, $1 \leq j \leq n$, $C[i,j]$ will contain the correct minimum cost of moving from row 1 to square (i,j), and $S[i,j]$ will contain the index of the column of the square from where we moved to square (i,j) in the last step along the optimal path to square (i,j).*

Proof: The correctness of the above procedure is based on the fact that it correctly implements the recursive definition given above. The base case is properly handled in Line 1 and 2, and the recursive case is properly handled in Lines 3 to 6. Note that since the loop defined in Line 3 goes from 1 to m, no element of C is accessed in either Line 5 or 6 before it has been computed. □

(4) Construct the Optimal Solution from the Computed Information. Consider the following piece of pseudocode, where C and S are the matrices computed above, m is the number of rows, and n is the number of columns.

```
Check-Sol(C, S, n)
1  j ← min_k{C[m, k]}
2  Print square (m, j)
3  for i ← m downto 1
4      Print square (i−1, S[i, j])
5      j ← S[i, j]
```

Claim 4 *The above procedure correctly outputs an optimal path from row m to row 1.*

Proof: Line 1 and 2 print the optimal square in row m. The first pass of the **for** loop will print the optimal square in row $m-1$. By setting $j \leftarrow S[i,j]$, the next pass though the **for** loop will print the optimal square in row $m-2$, and so on. □

(5) Running Time and Space Requirements. The CHECK procedure runs in $\Theta(mn)$ due to the nested loops (Lines 3 and 4), and it uses $\Theta(mn)$ additional space in the form of the C and S matrices. The CHECK-SOL procedure runs in time $\Theta(m)$ due to the loop in Line 3. It uses no additional space beyond the inputs given. Thus, the total running time is $\Theta(mn)$ and the total space requirement is $\Theta(mn)$.