Oliver Solorzano
CS 325
26 Jan 2020

1. What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?

Dynamic programming and divide-and-conquer are similar in that they both take a large problem and turn it into multiple sub-problems, which are then solved. They differ in the manner which they handle their sub-problems. For divide-and-conquer, the sub-problems are all independent of one another. When the sub-problems are solved, the solutions are combined to solve the original, whole problem. For Dynamic programming, the sub-problems overlap with one another. Since there is overlap, the same subproblems are usually calculated multiple times recursively. To prevent this, dynamic programming stores the answers to previously solved sup-problems to prevent repetition.

2. Shortest path counting: A chess rook can move horizontally or vertically to any square in the same row or the same column of a chessboard. Find the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner. The length of a path is measured by the number of squares it passes through, including the first and the last squares. Solve the problem:

   a) by a dynamic programming algorithm.

A chessboard can resemble a matrix that stores the values of a dynamic programming problem. In this case, an 8-by-8 matrix. Each square will hold the number of possible paths a rook can take from the corner of the board to that location (assuming a specific corner). We can let P(i, j) be the number of shortest paths from index (1, 1), where i and j can be any value from 1 to 8 (min, max moves in each direction on the board). For the edges adjacent to the corner the rook starts in*, there is only 1 possible path they can take, so 1 would fill out the indices in the adjacent rows. Ex:

1							
1							
1							
1							
1							
1							
1							
1*	1	1	1	1	1	1	1

We can work in a bottom-up approach, filling out the number of shortest possible paths the rook can take to a particular index, by using the values of previous indices. We can write the recurrence as:

$P(i,1) = 1$, for $1 <= i <= 8$

$P(1,j) = 1$, for $1 <= j <= 8$

$P(i,j) = P(i, j-1) + P(i-1, j)$ for $1 <= i$ or $j <= 8$

Using this recurrence equation the matrix can be filled as follows:

Oliver Solorzano
CS 325
26 Jan 2020

1	8	36	120	330	792	1716	3432*
1	7	28	84	210	462	924	1716
1	6	21	56	126	252	462	792
1	5	15	35	70	126	210	330
1	4	10	20	35	56	84	120
1	3	6	10	15	21	28	36
1	2	3	4	5	6	7	8
1*	1	1	1	1	1	1	1

*This is the number of shortest possible paths a rook can take to the opposite corner of the board by only moving vertically or horizontally.

b) by using elementary combinatorics.

Because we are limited by movement (either move horizontally or vertically), there is a limit to the number of possible moves a rook can make either upwards or horizontally toward the opposite corner. No matter what path is taken, the rook takes exactly 14 moves (squares) to get to the opposite corner (in any up/horizontal fashion). We know that 7 moves are vertical and 7 are horizontal, so we can assume that whatever combination of movement is taken in a shortest path, those 7 moves happen for each. Using combinatorics we can calculate the number of shortest possible paths by calculating 14-choose-7, or C(14, 7). This comes out to be C(14, 7) = 3432, which is the same answer we got by using the recurrence relation and filling out a matrix.

3. Maximum square submatrix: Given an $m \times n$ Boolean matrix B, find its largest square submatrix whose elements are all zeros. Design a dynamic programming algorithm and indicate its time efficiency. (The algorithm may be useful for, say, finding the largest free square area on a computer screen or for selecting a construction site.)

We can approach this problem in a similar fashion to other dynamic programming problems, by creating a matrix that stores representative values in a bottom-up fashion that allows us to find a max. We can track this by mirroring the positions in our input matrix to the corresponding values in our answer matrix. We can fill the first row and column of our answer matrix, “A”, based on the values of our input matrix. If we come across a “0” in our input, we set that value to 1if not we set it to zero. We can do so as follows:

```
*For (i = 0 to m) {

if B[i][0] == 0 {

        A[i][0] = 1;

}

else

A[i][0] = 0;
```

Oliver Solorzano
CS 325
26 Jan 2020

}

*Repeat for j, but modify for column instead

Once the initial row and column of the answer matrix are filled, we can move along the input matrix and add to our answer matrix based on the indices and the values. Since we are looking for squares only, and for values of zero only, we can keep adding value in our answer matrix to find a max, by adding on top of previous values. If we come across a "0" in our input matrix, we'll analyze the values in our answer matrix corresponding to the left/bottom/corner (a square). If we come across anything that isn't a "0", then we just set that index in our answer matrix to 0 (no chance of being part of a square, so zero value). We can do so by the following function:

if B[i,j] = 0 {

then take the min value of A[i][j-1], A[i-1][j-1], and A-i-1][j] and add 1

set A[i][j] equal to that sum

}

else {

set A[i][j] = 0

}

The algorithm would work as follows in this example:

Matrix "B" – problem input

7	0	6	7	0	0
0	4	0	0	0	0
0	0	0	0	0	0
5	0	0	0	0	6
6	0	8	9	0	0
0	0	7	0	0	0

Matrix "A" - answer

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

For this example the largest square has a column/row length value of "3". If we store the location of this max, in this case A[3][4], then we can backtrack to find the full region of the input matrix with values of only "0". Since we are going through the entire matrix to analyze, our time complexity is O(mn) – the size of the input matrix.

4. Consider the following instance of the knapsack problem with capacity W = 6.

Oliver Solorzano
CS 325
26 Jan 2020

Item	Weight	Value
1	3	$25
2	2	$20
3	1	$15
4	4	$40
5	5	$50

a) Apply the bottom-up dynamic programming algorithm to that instance.

Filling out a matrix using the bottom-up approach we would get:

	i=0	i=1	i=2	i=3	i=4	i=5
W=0	0	0	0	0	0	0
W=1	0	0	0	15	15	15
W=2	0	0	20	20	20	20
W=3	0	25	25	35	35	35
W=4	0	25	25	25	40	40
W=5	0	25	45	45	55	55
W=6	0	25	45	60	60	65

b) How many different optimal subsets does the instance of part (a) have?

There is only one optimal subset, using items #5 and #3. This is the only combination that gives us $65.

c) In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem's instance?

You can look to see if there is another combination in the table that results in the same max value (in this case $65). Since there is no other combination that gives us this max value, there is only one optimal subset for this case.

d) ***See code submitted to TEACH***

e) For the bottom-up dynamic programming algorithm, prove that its time efficiency is in $\Theta(nW)$, its space efficiency is in $\Theta(nW)$ and the time needed to find the composition of an optimal subset from a filled dynamic programming table is in $O(n)$.

For the knapsack function in my program, I have two loops, one for items and the other for the weights. The outer loop runs "n" times (for the number of items) and the inner loop runs "capacity" times (for the weights). The work in each loop runs in constant time so the time complexity would be the number of items times the possible weights, or O(n * capacity). Since we have to create an array to hold the values, the space complexity is the number of items times the number of weights, or O(n * capacity). The time needed to find the optimal subset is O(n) because the table has to be traversed to find the subset using the values (backtracing).