CSCI 270

Discussion Week 8

Matrix Chain Multiplication

If a chain of matrices is given, we have to find the minimum number of the correct sequence of matrices to multiply.

$A = A_1.A_2.\ldots\ldots A_n$

Operations to multiply an **mxn** matrix with a **nxk** matrix : O(mnk)

Example:

A = B.C.D

B is 2x10, C is 10x50, D is 50x20

B.(C.D) :

2x10x20 + 10x50x20 = 10400

(B.C).D :

2x10x50 + 2x50x20 = 3000

Using DP

OPT(i,j) = opt. cost to multiply matrices i through j

OPT(i,j) = Min { OPT(i,k) + OPT(k+1,j) + $R_i C_k C_j$ } for all k where $i \leq k < j$

Pseudocode

```
for i = 1 to n   OPT(i,i) = 0
for r = 1 to n-1
        for i = 1 to n-r
                j = i+r
                Use recurrence formula
        endfor
endfor
```

Time Complexity : $O(n^3)$

Problem

When their respective sport is not in season, USC's student-athletes are very involved in their community, helping people and spreading goodwill for the school. Unfortunately, NCAA regulations limit each student-athlete to at most one community service project per semester, so the athletic department is not always able to help every deserving charity. For the upcoming semester, we have S student-athletes who want to volunteer their time, and B buses to help get them between campus and the location of their volunteering. There are F projects under consideration; project i requires s_i student athletes and b_i buses to accomplish, and will generate $g_i > 0$ units of goodwill for the university. Our goal is to maximize the goodwill generated for the university subject to these constraints. Note that each project must be undertaken entirely or not done at all -- we cannot choose, for example, to do half of project i to get half of g_i goodwill.

Solution

This is very similar to the 0/1 knapsack problem except that we have two types of resources (students and buses) instead of one resource.

Value of the optimal solution for unique subproblems can be described as: OPT(i, S, B) = Maximum goodwill that can be generated using projects 1 to i, with S students and B buses.

The recurrence formula will be:

$OPT(i, S, B) = Max (g_i + OPT(i-1, S- s_i, B- b_i) , OPT(i-1, S, B))$

Initialization: OPT(0, S, B) and OPT(i, 0, B) and OPT(i, S, 0) can be set to zero since there will no goodwill if there are no projects or students or buses.

```
For i = 1 to F
    For j= 1 to S
        For k= 1 to B
            If ( s_i > j or b_i > k) then
             OPT(i, j, k) = OPT(i-1, j, k)
            Else
                OPT(i, j, k) = Max ( g_i + OPT(i-1, j- s_i, k- b_i) , OPT(i-1, j, k))
            Endif
        Endfor
    Endfor
Endfor
```

OPT(F, S, B) will hold the value of the optimal solution for the whole problem

To find the set of projects that we should select, we can then go top down (and print the list of projects selected):

```
j = S
k = B
For i= F to 1 by -1
        If g_i + OPT(i-1, j- s_i, k- b_i) > OPT(i-1, j, k) then
                Print i
                j = j- si
                k = k- bi
        Endif
Endfor
```

Complexity of the bottom up solution is O(FSB), which is pseudo-polynomial because S and B represent numerical values of the input terms rather than the size of the input.

The top down pass takes O(F) time. So, the whole solution will take O(FSB) time.

Problem

Suppose you are organizing a company party. The corporation has a hierarchical ranking structure; that is, the CEO is the root node of the hierarchy tree, and the CEO's immediate subordinates are the children of the root node, and so on in this fashion. To keep the party fun for all involved, you will not invite any employee whose immediate superior is invited. Each employee j has a value v_j (a positive integer), representing how enjoyable their presence would be at the party. Our goal is to determine which employees to invite, subject to these constraints, to maximize the total value of invitees.

Solution

We define the value of an optimal solution for a unique subproblem as:
OPT(i) = maximum “enjoyability value”/”fun value” of the subtree rooted at node i

Recurrence formula:

$OPT(i) = \text{Max} (vi + \Sigma_{g \in gi} OPT(g) , \Sigma_{c \in ci} OPT(c))$

Where g_i are the grandchildren of i and c_i are the children of i.

We store the OPT values at the corresponding nodes in the company hierarchy tree. So there will be no need to create any external arrays to hold OPT.

Bottom up pass:

At every node i we compute and send up (to the parent node) the following two values:

- $OPT(i) = \text{Max}\left(v_i + \sum_{g \in gi} OPT(g) , \sum_{c \in ci} OPT(c) \right)$
- $\sum_{c \in ci} OPT(c)$ (or zero if i has no children)

Note: we send up the value $\sum_{c \in ci} OPT(c)$, so that the parent node can easily compute the sum $\sum_{g \in gi} OPT(g)$ since the children of i are the grandchildren of I's parent node.

This pass takes linear time with respect to the size of the tree since each node is examined only once and the total number of additions performed is O(n).

To find out who will be invited to the party we go top down.

Invite (i)

If $vi + \Sigma_{g \in gi} OPT(g) > \Sigma_{c \in ci} OPT(c)$ Then

Print i

Call Invite with all g ϵ gi

Else

Call Invite with all c ϵ ci

Endif

The top down pass will take O(n) time since it will only be called a maximum of n times and the work inside the function is constant time (since the two sums $vi + \Sigma_{g \in gi} OPT(g)$ and $\Sigma_{c \in ci} OPT(c)$ can be stored at each node during the bottom up pass.)

Problem

You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Solution

We will first remove the complexity related to the rotation of the boxes by creating 3n box types as follows.
Given a box with sides measuring X < Y < Z, we will create 3 box types:

Width	Depth	Height
X	Y	Z
Y	Z	X
X	Z	Y

Now each of the 3n box types can only be used once in the stack. We then sort the 3n boxes based on decreasing base area (Width X Depth).
We define the value of an optimal solution for a unique subproblem as:
OPT(i) = Maximum height of a stack of boxes with box i at the top.

Recurrence formula will be:

$OPT(i) = h(i) + \text{Max}_{0<j<i \text{ and } i \text{ can fit on top of } j} (OPT(j))$

Note: we assume that Max will return a value of zero if we don't find a compatible box that i can fit on top of.

For i = 1 to 3n

$OPT(i) = h(i) + \text{Max}_{0<j<i \text{ and } i \text{ can fit on top of } j} (OPT(j))$

Base(i) = box no. corresponding to the value $\text{Max}_{0<j<i \text{ and } i \text{ can fit on top of } j} (OPT(j))$

Endfor

Note: The maximum value will not be necessarily at OP(3n). So we need to perform a linear search on OPT to find where the maximum value appears. We are also saving the id of the box we are putting i on, to simplify the top down pass. If box i does not fit on top of any other boxes, then Base(i) = 0

To find the boxes that will be going into the highest stack we can start with the box that has the maximum OPT value. Let's call that box i.

```
Print i
While Base(i) > 0
        i = Base (i)
        Print i
Endwhile
```

The bottom up will take $O(n^2)$ since at each of the 3n iterations we need to find the maximum of up to 3n terms.

The top down pass will only take O(n) time because we had saved the Base array in the bottom up pass. Otherwise, if this were not saved the top down pass would also take $O(n^2)$ time.

Problem

You are given an m × n binary matrix $g \in \{0, 1\}^{m \times n}$. Each cell either contains a “0” or a “1”. Give an efficient algorithm that takes the binary matrix g, and returns the largest side length of a square that only contains 1’s. You are **not** required to give the optimal solution.

a. Define (in plain English) subproblems to be solved.

Solution part a

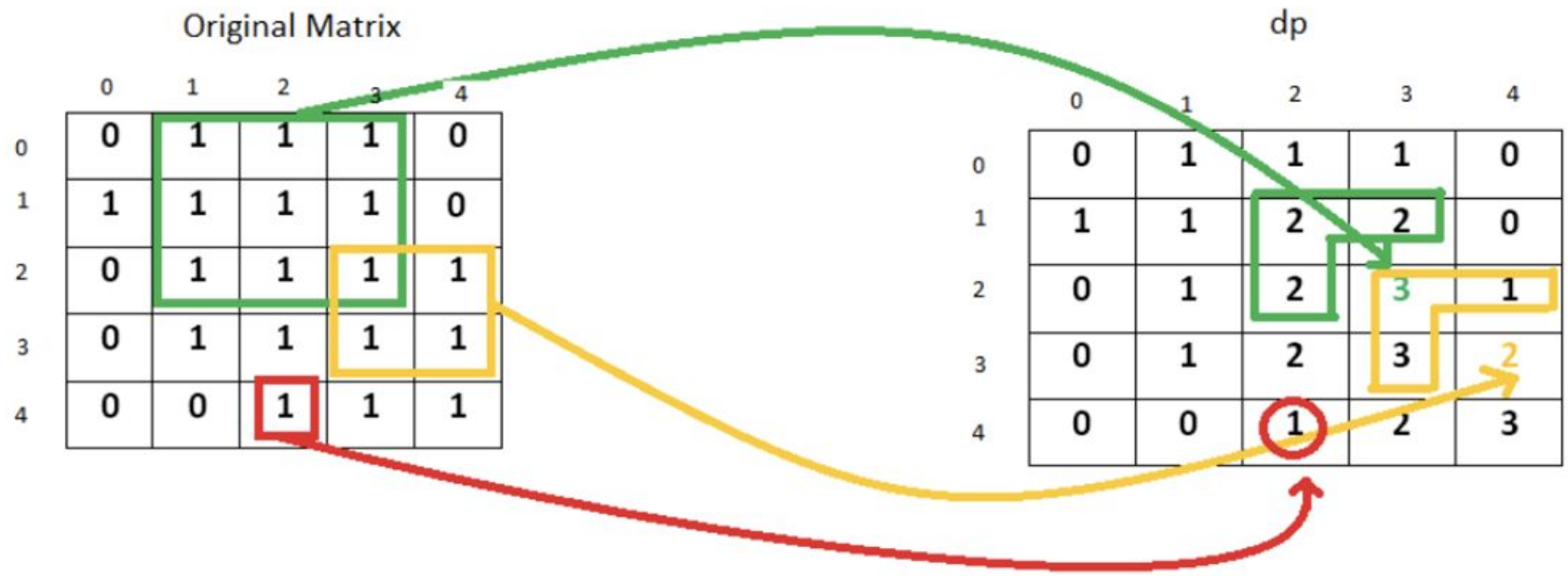
We define the m× n dynamic programming matrix OPT with OPT(i, j) being the side length of the maximum square matrix of 1’s whose right bottom cell is (i, j). That is, the sub matrix whose left top cell is (i − OPT(i, j) + 1, j − OPT(i, j) + 1) is the largest square matrix that is filled with 1’s and ending at cell (i, j), and either OPT(i, j) = min {i, j} or the matrix (i − OPT(i, j), j − OPT(i, j)) to (i, j) contains some 0’s.

b. Write a recurrence relation for the subproblems.

Starting from cell (1, 1), we compute the dynamic programming matrix f at cell (i, j) as

$OPT(i,j) = \min \{OPT(i-1, j), OPT(i-1, j-1), OPT(i, j-1)\} + 1, \quad i \geq 2 \ \& \ j \geq 2 \ \& \ g(i, j) = 1$

$g(i, j)$, Otherwise.


Original Matrix
dp
0 1 2 3 4
0 0 1 1 1 0
1 1 1 1 1 0
2 0 1 1 1 1
3 0 1 1 1 1
4 0 0 1 1 1
0 1 2 3 4
0 0 1 1 1 0
1 1 1 2 2 0
2 0 1 2 3 1
3 0 1 2 3 2
4 0 0 1 2 3

Pseudocode

set OPT(i, j) = 0 for all cells (i, j) that $i \in 1, \ldots, m$, $j \in 1, \ldots, n$. Set Ans = 0 for answer.

```
for i = 1, . . . , m do
    for j = 1, . . . , n do
        if g(i, j) = 0 or i = 1 or j = 1 then          ▷ boundary condition
            OPT(i, j) = g(i, j)
        else
            OPT(i, j) = min {OPT(i − 1, j), OPT(i, j − 1), OPT(i − 1, j − 1)}
    + 1
        if OPT(i, j) > Ans then                        ▷ update answer
            Ans = OPT(i, j)
```

Return: the largest side length stored in Ans.

The algorithm runs in O(mn) since there are at most O(mn) entries and each entry takes O(1) time to compute.

Reducing the space complexity:

For calculating OPTof i-th row we are using only the previous element and the (i − 1)-th row. Therefore, we don’t need 2D OPT matrix as 1D OPT array will be sufficient for this. Initially the OPT array contains all 0’s. As we scan the elements of the original matrix across a row, we keep on updating the dp array as per the equation OPT[j] = min(OPT[j − 1], OPT[j], prev), where prev refers to the old OPT[j-1]. For every row, we repeat the same process and update in the same dp array.