Theory Assignment-1: ADA Winter-2023

Chaitanya Arora (2021033) Raghav Sakhuja (2021274)

---

1 Question

Consider the problem of putting L-shaped tiles (L-shaped consisting of three squares) in an n × n square-board. You can assume that n is a power of 2. Suppose that one square of this board is defective and tiles cannot be put in that square. Also, two L-shaped tiles cannot intersect each other. Describe an algorithm that computes a proper tiling of the board. Justify the running time of your algorithm.

1.1 Answer

To solve this question we use the divide and conquer paradigm, in which we shall divide the nxn board into 4 quadrants and until we reach a small enough problem that can be solved without further division.

1.1.1 Sub-problems

We know that n is multiple of 2, so we can write $n = 2^t$, where t is a natural number. Let the columns and rows be number from 1 to 2^{t-1} and the defective squares be (x,y). We divide the board into 4 boards of equal size $n' = 2^{t-1}$. We define $fill(x, y, row_start, column_start, size)$. Here, if (x,y) belong to the quadrant we are currently working in we get the defective square for that quadrant. Else, we define the defective square as the corner of the quadrant which is towards (x,y). This continues on till we find a board of size 4x4. Here we can fit in a L shaped piece directly.

1.1.2 Combination of sub-problems

After every division, we get back 4 boards which have been filled with one defective piece in all of them. Now, one of them must contain the defective piece, while the remaining 3 would have an empty corner. Due to the way we divided them, these corners should meet in the center to form an L shaped cavity which can be filled with an L piece. We are proposing that an L shaped cavity can be filled with a single L shaped tile in $O(1)$ time. Since we are passing by reference, we do not have to spend any time combining the matrices.

1.1.3 Pseudo-code

Algorithm 1 Divide and Conquer L's

```
1: function FILL(x, y, i, j, size, board[1...n][1...n])
2:     if size == 2 then
3:         if (x, y) = (i, j) then
4:             Insert ⅃ at (i,j+1)(i+1,j)(i+1,j+1)
5:         end if
6:         if (x, y) = (i + 1, j) then
7:             Insert L at (i,j)(i,j+1)(i+1,j+1)
8:         end if
9:         if (x, y) = (i, j + 1) then
10:            Insert ⅂ at (i,j)(i+1,j)(i+1,j+1)
11:        end if
12:        if (x, y) = (i + 1, j + 1) then
13:            Insert Γ at (i,j)(i+1,j)(i,j+1)
14:        end if
15:    end if
16:    if Presence(x, y, i, j, size/2) then
17:        FILL(x, y, i, j, size/2, board[1...n][1...n])
18:        FILL(i + size/2 + 1, j + size/2 − 1, i + size/2, j, size/2, board[1...n][1...n])
19:        FILL(i + size/2 − 1, j + size/2 + 1, i, j + size/2, size/2, board[1...n][1...n])
20:        FILL(i + size/2 + 1, j + size/2 + 1, i + size/2, j + size/2, size/2, board[1...n][1...n])
21:        Insert ⅃ at (i,j+size/2)(i+size/2,j)(i+size/2,j+size/2)
22:    end if
23:    if Presence(x, y, i + size/2, j, size/2) then
24:        FILL(i + size/2 − 1, j + size/2 − 1, i, j, size/2, board[1...n][1...n])
25:        FILL(x, y, i + size/2, j, size/2, board[1...n][1...n])
26:        FILL(i + size/2 − 1, j + size/2 + 1, i, j + size/2, size/2, board[1...n][1...n])
27:        FILL(i + size/2 + 1, j + size/2 + 1, i + size/2, j + size/2, size/2, board[1...n][1...n])
28:        Insert L at (i,j)(i,j+size/2)(i+size/2,j+size/2)
29:    end if
30:    if Presence(x, y, i, j + size/2, size/2) then
31:        FILL(i + size/2 − 1, j + size/2 − 1, i, j, size/2, board[1...n][1...n])
32:        FILL(i + size/2 + 1, j + size/2 − 1, i + size/2, j, size/2, board[1...n][1...n])
33:        FILL(i + x, y, i, j + size/2, size/2, board[1...n][1...n])
34:        FILL(i + size/2 + 1, j + size/2 + 1, i + size/2, j + size/2, size/2board[1...n][1...n])
35:        Insert ⅂ at (i,j)(i+size/2,j)(i+size/2,j+size/2)
36:    end if
37:    if Presence(x, y, i + size/2, j + size/2, size/2) then
38:        FILL(i + size/2 − 1, j + size/2 − 1, i, j, size/2, board[1...n][1...n])
39:        FILL(i + size/2 + 1, j + size/2 − 1, i + size/2, j, size/2, board[1...n][1...n])
40:        FILL(i + size/2 − 1, j + size/2 + 1, i, j + size/2, size/2, board[1...n][1...n])
41:        FILL(x, y, i + size/2, j + size/2, size/2, board[1...n][1...n])
42:        Insert Γ at (i,j)(i+size/2,j)(i,j+size/2)
43:    end if
44: end function
45: function PRESENCE(x,y,i,j,size)
46:    if i < x < i + size and j < y < j + size then
47:        return true
48:    end if
49:    return False
50: end function
```

1.1.4 Final Solution

fill(i,j,0,0,n)

1.1.5 Run-time

We solve this problem by dividing the given problem of n^2 size into 4 problems of $(n/2)^2 = n^2/4$ size. We are able to combine these sub-problems in constant time as we already have the coordinates of the empty squares for each sub-problem and can insert an L piece in it.

Thus the recurrence relation is

$$T(n^2) = 4T(n^2/4) + O(1)$$

$$T(n^2) = 4^k T(n^2/4^k) + k.O(1)$$

let

$$n^2/4^k = 1$$

$$n^2 = 4^k$$

$$log_4(n^2) = k$$

thus we can write

$$T(n^2) = 4^{log_4(n^2)} T(1) + log_4(n^2)$$

$$T(n^2) = n^2 + log_4(n^2)$$

$$\Rightarrow O(n^2)$$

Thus we can compute the proper tiling of the in $O(n^2)$ using this Divide and Conquer algorithm.

2 Question

Suppose we are given a set L of n line segments in 2D plane. Each line segment has one endpoint on the line y = 0, one endpoint on the line y = 1 and all the 2n points are distinct. Give an algorithm that uses dynamic programming and computes a largest subset of L of which every pair of segments intersects each other. You must also give a justification why your algorithm works correctly.

2.1 Answer

This problem could be solved using Dynamic Programming by breaking it down into sub-problems, solving the optimal solution for each sub-problem, and reporting the final answer. We would consider an ideal subset to be the one in which the condition of the problem statement is achieved; that is, it would be a subset Of L of which every pair of segments intersects each other.

2.1.1 Pre-Processing State

We would be initializing an array(DP) of size n with initial value 1 as this would be the least possible size of the largest subset that would be an ideal subset.

We would also be sorting the pair of x_0 and x_1 along x_0 lines.

2.1.2 Sub-Problem

We would define our sub-problem as follows: We need to find the ideal subsets, in which if we were to add ith line segment, the initial condition of the ideal subset would still hold. This would give us the set containing the maximum number of lines which are pairwise intersecting in such a way that it is composed of only the first $(i-1)$ lines and definitely contains i^th line.

2.1.3 Recurrence

The recurrence relation for the dynamic programming approach can be derived as follows:

Let dp[i] denote the maximum size of the intersecting set that ends at index i. Then, for each line segment i, we iterate over all previously computed ideal subsets j (0 ¡= j ¡ i) and check if the ideal subset condition is satisfied between i and j. If it is, then we can extend the intersecting set that ends with the jth line segment by adding the i^th line segment to it, resulting in a larger intersecting set that ends with the i^th line segment.

Mathematically, the recurrence relation can be expressed as:

dp[i] = max(dp[j] + 1) for all j such that 0 j ¡ i and Intersect(i, j, x0[1..n], x1[1..n]) is true.

We then take the maximum of all such sizes and assign it to dp[i].

Here Intersect(i, j, x0[1..n], x1[1..n]) is a function that checks whether the intervals represented by subsets of line segments x intersect or not.

2.1.4 Solution of Sub-Problem

The solution to our subproblem can be solved by iterating through all previously computed ideal subsets and then increasing and incrementing the size of the subset by one which when the ith line is added still remains the ideal subset. If we could ensure that the subset would remain ideal even after adding this line segment to that subset we would be increasing its size by one, then We then take the maximum of all such sizes of ideal subsets and assign it to dp[i].

2.1.5 Pseudo-code

Algorithm 2 Max Intersecting Set

Input: $x_0[1..n]$ and $x_1[1...n]$ arrays of size n
Initialize dp array with dimensions n+1 with all values set to 1.
Set $dp[0] = 0$

```
1: function INTERSECT(i, j, x0[1..n], x1[1..n])
2:     if (x1[j] − x0[j]) ∗ (x0[i] − x0[i]) > 0 then
3:         return true
4:     end if
5:     return false
6: end function
7: for i ← 0ton − 1 do
8:     for j ← 0toi − 1 do:
9:         if INTERSECT(i, j, x0[1..n], x1[1..n]) then
10:            dp[i] ← max(dp[j] + 1, dp[i])
11:        end if
12:    end for
13: end for
```

2.1.6 Final Solution

Our final solution could be computed by iterating through the entire DP array and returning the maximum size of the ideal subset.

2.1.7 Run-time

The time complexity of the given code is $O(n^2)$, where n is the length of the input arrays x0 and x1. The pre-processing state would take $O(nlogn)$ using the merge sort algorithm already known to us. This is because the code has a nested loop structure, where the outer loop runs for n times, and the inner loop also runs for n times in total over all iterations of the outer loop. Within the inner loop, the Intersect function takes constant time O(1) to execute.

Therefore, the total time complexity of the code is $O(n^2)$.

3 Question

Suppose that an equipment manufacturing company manufactures s_i units in the i^{th} week. Each week's production has to be shipped by the end of that week. Every week, one of the three shipping agents A, B and C are involved in shipping that week's production and they charge in the following:

• Company A charges a rupees per unit.

• Company B charges b rupees per week (irrespective of the number of units), but will only ship for a block of 3 consecutive weeks.

• Company C charges c rupees per unit but returns a reward of d rupees per week, but will not ship for a block of more than 2 consecutive weeks. It means that if s_i unit is shipped in the i^{th} week through company C, then the cost for i^{th} week will be $cs_i - d$.

The total cost of the schedule is the total cost to be paid to the agents. If s_i unit is produced in the i^{th} week, then s_i unit has to be shipped in the i^{th} week. Then, give an efficient algorithm that computes a schedule of minimum cost.

3.1 Answer

To solve this problem, we came up with a Dynamic Programming solution that uses the concept of finding the next optimal solution, which in this case would be the minimum cost to ship the entire week's production by optimally choosing the shippers.

3.1.1 Pre-Processing State

There is no pre-processing state in the way we approach this problem. Our approach is a completely hands-down approach.

3.1.2 Sub-Problem

We would define the sub-problem in the following way: We know the minimal price to ship all the units till $i - 1^{th}$ week. Now we want to find out the minimum cost of shipping the production up to i^th week using any of the three shipping agents in the i^th week.

3.1.3 Recurrence

The recurrence relation for the sub-problem defined in the previous section would be:- Presuming that we have calculated and stored the optimal solutions of all the previous states till now. We can calculate the optimal solution for the current state by three simple conditions.

1. While considering the possibility that Shipper A would provide the minimum cost of shipment for this week, our solution would depend on the solution of the sub-problem of the previous week. The reason that we don't have to handle any special cases or sub-cases for this part of the solution is that there are no restrictions as to how many times this shipper could be used consecutively and there is no cool-down period for the same.

2. While considering the possibility that Shipper B would provide the minimum cost of shipment for the current week, our solution would depend on the solution of the sub-problem of three weeks earlier. This is because Shipper B only ships for a period of 3 weeks consecutively. Therefore we don't need the optimal solutions for the previous week or the week before the previous week, and still, we would be able to find the optimal solution for this week.

3. While considering the possibility that Shipper C would provide the minimum cost of shipment for the current week, our solution would depend on two things:-

3.1. Primarily, the solution would depend on the optimal solution of the previous week.

3.2. Secondly, we would have to check whether we can ship the week's unit using Shipper C. This is because there is a cool-down period for using the Shipper C after using it for shipment for two consecutive weeks. Therefore, in this case, our solution would depend on the shippers used to Ship the units of the previous two weeks.

Although once we know the Shippers of the last two weeks, it's easy to decide whether we have to consider the possibility of shipping the units with Shipper C or not.

We have also kept a track of the number of times we have consecutively used the Shipper C to transport our units and this is achieved by storing 3 possible DP states in the DP array by giving it a 2nd Dimension, wherein we have stored not using the C shipper in the 1st row, using the shipper C for the first time in 2nd row and using the shipper C for the 2nd time consecutively in the 3rd row.

3.1.4 Solution of Sub-Problem

The sub-problem that solves the original problem is to find the minimum cost of shipping the production up to the last week using any of the three shipping agents.

The Dp[i][0] gives us the minimal cost of transporting the units without the use of shipper C on i^th day.

The Dp[i][1] gives us the minimal cost of transporting the units using shipper C for the first time after using some other shipper previously on i^th day.

The Dp[i][2] gives us the minimal cost of transporting the units using shipper C for the second time in consecutive orders on i^th day.

The base case of the DP solution would be the following:

$DP(0)(0) = 0$

$DP(0)(1) = 0$

$DP(0)(2) = 0$

3.1.5 Pseudo-code

Algorithm 3 Min Cost For Shipment

Input: n, a, b, c, d, and an array s of size n

Initialize dp array with dimensions (n+1) x 3, with all values set to positive infinity.

Set $dp[0][0] = dp[0][1] = dp[0][2] = 0$

for $i \leftarrow 1ton + 1$ **do**
  $av \leftarrow min(dp[i-1][0], dp[i-1][1], dp[i-1][2]) + a * s[i-1]$
  if $i >= 3$ **then**
    $av \leftarrow min(dp[i-3][0], dp[i-13][1], dp[i-3][2]) + 3 * b$
  end if
  $dp[i][0] = min(av, bv)$
  $dp[i][1] = dp[i-1][0] + c * s[i-1] - d$
  $dp[i][2] = dp[i-1][1] + c * s[i-1] - d$
end for
return $min(dp[n][0], dp[n][1], dp[n][2])$

3.1.6 Final Solution

We can report the final solution by taking the maximum of DP[n][0], DP[n][1] and DP[n][2].

3.1.7 Run-time

We solved this problem using an iterative DP approach and the following is the justification for our Time Complexity:

We are able to solve this problem by creating a 2d array of dimensions $3 * (n + 1)$.

Thus the Time Complexity Analysis of our solution can be defined by the following:

For the for loop: $O(n)$. Inside the for loop we are computing 3 states of DP. This takes a time of $O(1)$. per state. This is because we are only accessing the items from the memoization table and that gives the results in $O(1)$ time complexity. Therefore in each iteration, we are computing the optimal solution for that step using $O(3)$ time complexity. Thus we can compute the optimal answer (the minimum cost of shipping our units) in $O(3 * n)$, this can be further represented as $O(n)$.