CS403: Algorithm Design and Analysis

Feb-Jun 2017

Indian Institute of Technology Mandi

Assignment 3

Priyansh Saxena
B14118
Computer Science and Engineering

CS 403 – Algorithm Design and Analysis

Assignment 3

Problem 1: Implement weighted-interval scheduling. The algorithm for the same is given below, in three functions:

Compute-Opt(j) {

If $j = 0$, then Return 0

Else Return max(v_j + Compute-Opt(p(j)), Compute-Opt (j – 1))

Endif

}

M-Compute-Opt(j) {

If $j = 0$, then Return 0

Else if M[j] is not empty, then Return M[j]

Else

M[j] = max(v_j + M-Compute-Opt (p(j)) , M-Compute-Opt (j – 1))

Return M[j]

Endif

}

Find-Solution(j) {

If $j = 0$ then

Output nothing

Else

If $v_j + M[p(j)] \geq M[j-1]$ then

Output j together with the result of Find-Solution(p(j))

Else

Output the result of Find-Solution(j – 1)

Endif

Endif

}

Let the value of the optimal solution of the problem of size 'n' be denoted by OPT(n). Now, as was done in the interval-scheduling problem, sort the tasks in increasing order of finish-time.

Now, consider the j^{th} task. We include this task into the schedule, if the sum of values of this task - v_j - and the optimal-value of the schedule preceding task 'j' and not conflicting with it – OPT(p(j)) – is at least as good as the optimal value till the $(j-1)^{th}$ task. Otherwise, we leave this task out of our schedule. Also, we update the optimal-value of the schedule till task j as OPT (j) = max(v_j + OPT (p(j)), OPT (j – 1)).

Sorting the tasks takes O(n.logn) time. Computation of conflict-intervals, p(j) takes $O(n^2)$ time. M-Compute-Opt will compute the optimal values in at most O(n) operations and then the Find-Solution calls itself recursively only on strictly smaller values, making a total of O(n) recursive calls spending constant time per call. Therefore, the algorithm has an overall time-complexity of $O(n^2)$.

The space-complexity of the memoisation-based solution is O(n).

Problem 2: Counting inversions in a list of values, as per some order.

The algorithm to count the number of inversions is a simple variation of the merge-sort algorithm. At each divide-step, we count the number of inversions in each half. At each merge-step, we count the number of inversions while merging the two-halves by adding the number of left-values in the 'smaller half' when an element from the 'bigger half ' is adjudged smaller.

Merge-and-Count(A , B) {
    Maintain a Current pointer into each list, initialized to point to the front elements
    Maintain a variable Count for the number of inversions, initialized to 0

    While both lists are nonempty:
        Let a i and b j be the elements pointed to by the Current pointer
        Append the smaller of these two to the output list
        If b_j is the smaller element then
            Increment Count by the number of elements remaining in A
        Endif
        Advance the Current pointer in the list from which the smaller element was selected.
    EndWhile

    Once one list is empty, append the remainder of the other list to the output
    Return Count and the merged list
}

Sort-and-Count(L) {

If the list has one element then

there are no inversions

Else

Divide the list into two halves:

A contains the first ceil(n/2) elements

B contains the remaining floor(n/2) elements

(r_A , A) = Sort-and-Count(A)

(r_B , B) = Sort-and-Count(B)

(r , L) = Merge-and-Count(A, B)

Endif

Return $r = r_A + r_B + r$, and the sorted list L

}

The space-complexity of the algorithm is O(n), as in Merge-Sort.

The time-complexity of the algorithm is O(n.logn). This can be shown with the following arguments.

1. Merge-and-Count procedure takes O(n) time.

2. Since there are logn divisions, there will (logn) merge operations.

Problem 3: Finding the Closest Pair of Points in a plane.

The algorithm used to solve the problem uses divide-and-conquer. First, sort the points on their x-coordinates, in ascending order. Next, find the closest-pair of points in the left half and right half of the plane separately. Let d be the minimum of the distances between points in these two pairs.

Now find the set of points which lie at a distance less than or equal to d from the half-line. Sort this set on y-coordinates. Next, search for the closest-pair of points in this sorted-list by considering every-pair of points. It can be shown that the points will lie within 15 positions of each other.

Recursively follow this solution for smaller sub-problems, using brute-force for less than four point in a set.

Closest-Pair(P) {

Construct P_x and P_y (O(n log n) time)

(p_0^*, p_1^*) = Closest-Pair-Rec(P_x , P_y)

}

Closest-Pair-Rec(P_x , P_y)

If $|P| \leq 3$ *then*

find closest pair by measuring all pairwise distances

Endif

Construct Q_x , Q_y , R_x , R_y *(O(n) time)*

(q_0^*, q_1^*) = *Closest-Pair-Rec(* Q_x , Q_y *)*

(r_0^*, r_1^*) = *Closest-Pair-Rec(* R_x , R_y *)*

$\delta = min(dist(q_0^*, q_1^*), dist(r_0^*, r_1^*))$

x^* = *maximum x -coordinate of a point in set Q*

$L = \{(x, y) : x = x^*\}$

S = points in P within distance δ *of L .*

Construct S_y *(O(n) time)*

For each point $s \in S_y$ *, compute distance from s to each of next 15 points in* S_y

Let s , s' be pair achieving minimum of these distances (O(n) time)

If $d(s, s') < \delta$ *then*

Return (s , s')

Else if $dist(q_0^*, q_1^*) < dist(r_0^*, r_1^*)$ *then*

Return (q_0^*, q_1^*)

Else

Return (r_0^*, r_1^*)

Endif

The time-complexity of the algorithm is O(n.logn).

1. The sorting of the points takes O(n.logn) time.
2. The remainder of the algorithm then divides the points in two equal halves and spends constant time to merge the solution from those two subproblems.

Therefore, the algorithm has a time-complexity of O(n.logn).

Problem 4: Segmented least-squares problem.

Suppose our data consists of a set P of n points in the plane, denoted (x_1, y_1), (x_2, y_2), . . . , (x_n, y_n); and suppose $x_1 < x_2 < . . . < x_n$. Given a line L defined by the equation $y = ax + b$, we say that the error of L with respect to P is the sum of its squared "distances" to the points in P:

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

The line of minimum error is y = ax + b, where

$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \quad \text{and} \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}.$$

Now, to compute all the errors in $O(n^2)$, we can apply a pre-computation step, by noting that each of the terms x_i, y_i, x_i^2, y_i^2 and x_iy_i can be computed in O(1) by cumulative addition. Therefore, it takes O(n) time in pre-computation, and $O(n^2)$ time for every pair (i,j) (since the results can be obtained in O(1) after pre-computation).

Set initial values:

$cumX_1 = X_1$
$cumY_1 = Y_1$
$cumXX_1 = X_1^2$
$cumYY_1 = Y_1^2$
$cumXY_1 = X_1Y_1$

For i in [2,n]

$cumX_i = cumX_{i-1} + X_i$
$cumY_i = cumY_{i-1} + Y_i$
$cumXX_i = cumXX_{i-1} + X_i^2$
$cumYY_i = cumYY_{i-1} + Y_i^2$
$cumXY_i = cumXY_{i-1} + X_iY_i$

For every pair (i, j), i < j

If i = 1 then

$X = cumX_j$
$Y = cumY_j$
$XY = cumXY_j$
$YY = cumYY_j$
$XX = cumXX_j$

Else

$X = cumX_j - cumX_{i-1}$
$Y = cumY_j - cumY_{i-1}$
$XY = cumXY_j - cumXY_{i-1}$
$YY = cumYY_j - cumYY_{i-1}$
$XX = cumXX_j - cumXX_{i-1}$

End If

Calculate 'a' and 'b' by replacing the summations with X, Y, XY, and XX at appropriate positions.

Calculate error by expanding the expression of error and replacing X, Y, XX, YY and XY using the above.