CS 180 Homework 5

Jiaping Zeng

11/20/2020

P1 Consider the divide and conquer algorithm for finding the closest pair of points. Analyze the time complexity of the algorithm. Include and discuss a detailed discussion of how to manage points in the x-dimension and how to manage and search points in the y-dimension.

Algorithm:

1. Sort the points by x-coordinate and split the list into halves by x.
2. Find the closest points recursively by repeating step 1 for each of the two halves, until there is only two points left, in which case the distance is simply the distance between those two points. Let m be our minimal distance found in this step.
3. Now we look at the rectangular area with width $2m$, centered at the middle line we found in step 1.
4. Sort the points within the rectangular area by their y-coordinate and split the list of points within the rectangular area into halves by y.
5. Now we perform steps 1-2 but halving by y-coordinate each time. If a new minimum distance is found, i.e. if $d < m$, set $m = d$.
6. Return m.

Proof of correctness: By contradiction. Suppose our algorithm does not return the closest pair of points, then there must exists another pair of points with a closer distance. Such points could either exist in the two x-halves or the center rectangular area. The first case is not possible since we have recursively checked all the points by their x-coordinate. The second case also not possible since we have recursively checked all the points in the center rectangular area by their y-coordinate. Therefore no such points exist and our algorithm is correct by contradiction.

Time complexity: $O(n \log n)$; sorting the points by x and finding the minimum takes $O(n \log n)$ time, then finding points within the center area takes $O(n)$ time, finally sorting the points by y and finding the minimum in the center rectangular area takes $O(n \log n)$ time. Therefore overall our algorithm is $O(n \log n)$.

6.4 (a) **Answer**: Let $M = 100$, $N = \{1, 2\}$ and $S = \{1, 1\}$. The algorithm would give us $[NY, SF], cost = 102$ when the optimal solution would be $[SF, SF], cost = 2$.

(b) **Answer**: Let $M = 1$, $N = \{1, 10, 1, 10\}$ and $S = \{10, 1, 10, 1\}$. Since moving cost is far lower than the cost difference each month, the optimal strategy is to move every month, i.e. $[NY, SF, NY, SF], cost = 7$. Not moving in any of the months would increase the cost by 9.

(c) **Algorithm**:

1. Let C_{SF} and C_{NY} be the cost functions of the optimal plan, where C_{SF} is the cost of an optimal plan that ends in SF whereas C_{NY} is cost of one that ends in NY. Set $C_{SF}(0) = C_{NY}(0) = 0$.
2. For each month m, calculate $C_{SF}(m)$ and $C_{NY}(m)$ as follows:
   - $C_{SF}(m) = \min\{C_{SF}(m-1), C_{NY}(m-1) + M\} + S_n$
   - $C_{NY}(m) = \min\{C_{SF}(m-1) + M, C_{NY}(m-1)\} + N_n$
3. Return $\min\{C_{SF}(n), C_{NY}(n)\}$.

Proof of correctness: By induction on n, the number of months.

Base case: $n = 0$, our algorithm returns 0 as expected.

Inductive step: Suppose our algorithm works for n months, we want to show that it will also work for $n + 1$ months. Since $m - 1$ is at most n, by inductive hypothesis, we will always have the correct values for $C_{SF}(m-1)$ and $C_{NY}(m-1)$.Then we have the correct arguments for min, which will give us the correct $C_{SF}(m)$ and $C_{NY}(m)$ values. Taking the min of the two gives us the expected result.

Therefore our algorithm is correct by induction.

Time complexity: $O(n)$; the algorithm runs constant time calculations for each of the n months.

6.6 **Algorithm**:

1. Let $s(j, k)$ be the function that returns the line slack using words $w_j, \ldots, w_k$, i.e. $s(j, k) = L - (\sum_{i=j}^{k-1}(c_i + 1) + c_k)$. Define $s(j, k) = \infty$ if the line length exceeds L.
2. For each word w_i in the words list, compute $s(i, j)$ for each word w_j after it.
2. Let $S(m)$ be the optimal solution function using the first m words. Set $S(0) = 0$ with a corresponding empty partition.
3. For each word w_k in the words list, calculate $S(k) = \min_{1 \le j \le k}\{s(j, k)^2 + S(j-1)\}$. Save the partition corresponding to such $S(k)$.
4. Return the partition corresponding to $S(n)$.

Proof of correctness: By induction on n, the number of words.

Base case: $n = 0$, then our algorithm returns the empty partition as expected.

Inductive step: Suppose our algorithm works for n words, we will show that it will also work for $n + 1$ words by showing that both loops will work as expected.

In the $s(j, k)$ loop, since our function traverses through every word in the list, adding a new word to the list means it will simply run one more iteration, which since we have defined it as desired, will give us the correct result. The inner loop will then also run an extra iteration for each word. Therefore our $s(j_k)$ calculations is correct.

The $S(m)$ loop will run an extra iteration and find the minimal slack using $S(j-1)$ in iteration j. Since $j-1$ is at most n, by inductive hypothesis $S(j-1)$ will always return the correct result. Then our algorithm will also give us the correct value here. Therefore our algorithm is correct by induction.
Time complexity: $O(n^2)$; computing all $s(i,j)$ takes $O(n^2)$ as it loops all words and performs $O(n)$ calculations for each word. Similarly, computing $S(k)$ also takes $O(n^2)$. Therefore the algorithm is $O(n^2)$ overall.

6.9 (a) Let $X = \{5,5,5,5,5\}$ and $S = \{4,1,1,1,1\}$, then the optimal solution would be reboot on days 2 and 4 which gives us a total of 12 terabytes processed over 5 days.

(b) **Algorithm**:

1. Let $T(i,j)$ be the number of terabytes processed from day i to day n, with the last reboot j days before day i. Set $T(n,\cdot) = \min\{x_n, s_n\}$.
2. For $1 \le i \le n$, do the following:
   - For $1 \le j \le i$, set $T(i,j) = \max\{T(n-i+1,1), T(n-i+1,j+1) + \min\{x_{n-i}, s_{n-i}\}\}$
3. Return $T(1,1)$.

Proof of correctness: By induction on n, the number of days.
Base case: $n = 1$, then our algorithm returns $\min\{x_1, s_1\}$ as expected.
Inductive step: Suppose our algorithm works for n days, we want to show that it will also work for $n+1$ days. By construction of our algorithm, we simply need to show that the first iteration works as expected and the rest are identical to the iterations for n days. For the first iteration, we are checking $T(n+1-i+1,1) \implies T(n-i+2)$, where argument is at most $n+1$ since $1 \le i$. Since we have defined $T(n+1)$ in step 1 and $T(n)$ and below all work by inductive hypothesis, our algorithm returns the right result.
Therefore our algorithm works by induction.
Time complexity: $O(n^2)$ since we are iterating through n days and performing $O(n)$ calculations for each iteration.

P5 Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown. You need to use dynamic programming to solve this problem.
Algorithm:

1. Let $S(n,X)$ be the number of ways to get sum X with n dice each with m faces. Set $S(1,\cdot) = 1$.
2. For the i-th die, do the following:
   - For $i \le j \le X$, set $S(i,j) = \sum_{k=1}^{m} S(i-1, j-k)$.
3. Return $S(n,X)$.

Proof of correctness: By induction on n, the number of dice.
Base case: $n = 1$, our algorithm returns 1 as expected.
Inductive step: Suppose our algorithm works for n dice, we want to show that it will also work for $n+1$ dice. For the $(n+1)$-th die, our algorithm sums up all the $S(n,\cdot)$ entries, which all work by

inductive hypothesis. Therefore it returns the correct result.

Therefore our algorithm is correct by induction.

Time complexity: $O(nmX)$, since we are using three nested loops with $O(n)$, $O(X)$ and $O(m)$ respectively from the outermost loop to the innermost.

P6 You are given a set of n types of rectangular 3D 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 2D base of the lower box are each strictly larger than those of the 2D 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.

Algorithm:

1. Calculate the area of the three different faces for each box, i.e. $A_{F_1} = w(i)h(i)$ (base area), $A_{F_2} = w(i)d(i)$, $A_{F_3} = h(i)d(i)$. Keep track of which box these areas belong to.
2. Duplicate each box twice, rotating it each time such that we have all three rotations of each box.
3. Let $H(m)$ be the height of the tallest stack with m types of boxes. Set $H(0) = 0$.
4. Sort the boxes from greatest base area (A_{F_1}, as defined above) to smallest.
5. For each box b_k in the sorted list, calculate $H(k) = \max_{1 \leq j \leq k}\{H(j-1) + h(i)\}$.
6. Return $H(3n)$, since step 2 gave us $3n$ boxes.

Proof of correctness: By induction on n, the number of boxes.

Base case: $n = 0$, then our algorithm returns $H(0) = 0$ as expected.

Inductive step: Suppose our algorithm returns the correct height for $3n$ types of boxes, we want to show that it will also work for $3n + 1$ types. Since $j - 1$ is at most $3n$ in our loop, by inductive hypothesis $H(j-1)$ will always give us the correct result. Then our algorithm will give us the correct argument for max, which will give us the tallest stack as expected.

Therefore our algorithm is correct by induction.

Time complexity: $O(n^2)$; sorting the boxes by base area takes $O(n \log n)$ time, looping through $3n$ boxes and check $O(n)$ boxes takes $O(n^2)$ time, so overall the algorithm is $O(n^2)$.