CS170 Fall 2013 Solutions to Homework 8

Zackery Field, section Di, 103, `cs170-fe`

November 1, 2013

1. 6.4 Reconstruction

You are given a string of n characters $s[1, \ldots, n]$, which you believe to be a corrupted text document in which all punctuation has vanished. You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function $dict(\cdot)$: for any string w,

(a) Give a dynamic programming algorithm that determines whether the string s can be reconstructed as a sequence of valid words. The running time should be at most $O(n^2)$, assuming that the calls to dict take unit time.

The smallest substring s_w that will yield $dict(w) = true$ is of length 1. If we are at the end of the string, assuming that the optimal solution to the rest of the string ($s[1, \ldots, n-1]$) has been determined, then the remaining step is to determine if the last character in the string is included in a word or not.

(1[# subproblems])There are n of these 'is s_i in a word or not' subproblems. And with each of these subproblems there is a resulting specification of which word s_i suffixes. (2[# choices])For each of these subproblems there are at most n possible words that s_i can suffix. (3[recurrence]):

$$DP[s_j] = \begin{cases} if(dict(s_{1,j}) = true) : return\ construction & \text{for a word } s_{1,j} \\ \forall_{1<i<j}(dict(s_{i,j}) = true : DP[s_i];) & \text{if } s_j \text{ is in a valid word} \\ no\ reconstruction & w \text{ is not in a valid word} \end{cases}$$

(4 [order]) The order with which you proceed from s_n to s_1 is dependent upon which choice of word is made, but you will make at most n checks on for each s_i with each check taking a unit time dictionary check. For a total of $O(n^2)$ checks.

Continued on Page 6

2. 6.14 Cutting Cloth

You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1, n]$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the proucdt is c_i. Assume that a_i, b_i, and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically . Design an efficient dynamic programming solution that determines the best strategy for cutting the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired. For the purposes of corect indexing, the top left point is $(0, 0)$ and the bottom left index is (X, Y).

There are X*Y maximum subproblems that must be computed because there is the possibility that for each coordinate in the cloth you will have to make a cut decision.

Starting at $(0, 0)$ we have 4 options for each of the possible cuts that we can make for a given product. These options are defined by the dimensions (a_i, b_i) of each piece p_i. For some piece p_i we can cut $(x + a_i, y)$, or $(x, y + a_i)$ since each piece can be rotated. For this same piece we can alternatively make the cuts $(x + b_i, y)$, or $(x, y + b_i)$. And the final choice is to not cut p_i at all, in which case we simply increment to the next possible piece. This makes for a total of $\Theta(4 * n)$ choices on the first subproblem.

Reccurrence relation for some coordinate (x, y) within the top left corner of a 'sub-box' of size (a', b'), and some piece $p_i = (a_i, b_i)$:

$$(x, y, a', b') = Max \begin{cases} c_i + (x + a_i, y, & b' = b_i, i + 1) \\ c_i + (x, y + a_i, & a' = b_i, i + 1) \\ c_i + (x + b_i, y, & b' = a_i, i + 1) \\ c_i + (x, y + b_i, & a' = a_i, i + 1) \\ 0 + (x, y, & i + 1) \end{cases}$$

In the case where there are no products where one of their dimensions is the exact size of the current 'sub-box' then choose any item cut.

The order of each cut is defined by the top left corner of each sub-box. With this ordering, once an optimal choice has been made for cut-site (i, j) you know that you will never return to that site. The maximum selling price will be the what is natually returned by the recurrence. To also get the list of products sold simply return when you reach $(i, j) = (X, Y)$ and pass back up each of the cuts made.

Continued on Page 7

3. 6.20 Optimal binary search trees

Suppose we know the frequency with which keywords occur in programs of a certain language. We want to organize them in a binary search tree, so that the keyword in the root is alphabetically bigger than all the keywords in the left subtree and smaller than all the keywords in the right subtree.

Figure 6.12 has a nicely-balanced example on the left. In this case, when a keyword is being looked up, the number of comparisons needed is at most three: for instance finding while, only the three nodes 'end', 'then', and 'while' get examined. But since we know the frequency with which keywords are accessed, we can use an even more fine-tuned cost function, the average number of comparisons to look up a word:

$$cost(\cdot) = 1(p_{1_1} + \ldots + p_{1_n}) + \cdots + n(p_{n_1} + \cdots + p_{n_n})$$

By this measure, the best search tree is the one on the right, which has a cost of 2.18.

Give an efficient algorithm for the following task:

Input: n words (in sorted order); frequencies of these words: $p_1, p_2, \ldots, p_n$.

Output: The binary search tree of lowest cost.

Arbitrarily select one of the n words to be the root r of the tree. The possible left and right subtrees of r are determined alphabetically. Let the set of words in the left subtree be called L, and the set of words in the right subtree be called R. Each of the subproblems is the determination of the minimizing choice of the roots of each of the subtrees. Let S be some alphabetically determined subset of all the words W, and d be the depth at which S will be rooted. For a word $w \in S$, where we assume that its optimal left and right subtrees are already known, we can use this recurrence to minimize cost.

$$DP[w_i] = Min \begin{cases} d * p_w + c_{unweighted} + c_{weighted} & \text{, include w as the root of } S \\ DP[w_{i+1}] & \text{, don't include w as the root of } S \end{cases}$$

Computing the inclusion of w is linear in time, but computing whether or not some other root minimizes the cost takes much longer. For some tree, we can use the cost function defined above to determine the weighted cost of the optimal subtree s to be $c_{weighted} = \sum_{i \in s} d_i * p_i$. Also, for the purposes of 'rerooting' the subtree, we will track the unweighted cost of all the words in the subtree as the quantity $c_{unweighted} = \sum_{i \in s} p_i$.

As you can see from the recurrence, when we choose w to be the root we must add the weighted cost of its two subtrees and the unweighted cost of the two subtrees (this takes care of the 'rerooting' of the trees), and we must also add the cost of w itself.

For each of the n choices of the root of the tree it takes $O(n^2)$ time to determine the optimal subtrees on each side of the root for a total running time of $O(n^3)$.

4. 6.29 Exon chaining

Each gene corresponds to a subregion of an overall genome (the DNA sequence); however , part of this region might be 'junk DNA'. Frequently, gene consists of several pieces called exons, seperated by junk fragments called introns. This complicates the process of identifying genes in a newly sequenced genome.

Suppose we have a new DNA sequence and we want to check whether a certain gene (a string) is present in it. Because we cannot hope that the gene will be a contiguous subsequence, we look for partial matches-fragments of DNA that are also present in the gene (actually, even these partial matches will be approximate, not perfect). We then attempt to assemble these fragments.

Let $x[1, \ldots, n]$ denote the DNA sequence. Each partial match can be represented by a triple (l_i, r_i, w_i), where $x[l_i, \ldots, r_i]$ is the fragment and w_i is the weight representing the strength of the match (it might be a local alignment score or some other statistical quantity). Many of these potential matches could be false, so the goal is to find a subset of the triples that are consistent (nonoverlapping) and have a maximum total weight.

Show how to do this efficiently.

Start at the last character in the string s_n and assuming that you have the optimal triple-set up to s_n. With this knowledge, we need to simply make the optimal choice for the triple that ends at s_n, or the triple (l, n, w). If there are m comparable subsequences then you can expect there to be at most m triples that satisfy the form (l, n, w). Starting at any s_k, letting $A(k) = [1, \ldots, k]$ we can define the recurrence:

$$A(k) = Max \begin{cases} A(k-1), \text{do not include } s_k \text{ in a match} \\ \forall_i r_i = k + A(i), \text{find the optimal match} \end{cases}$$

In order to reduce the running time of the search process we can preprocess the values of the triples for a given (l', r') we can keep the one with the greatest weight w. This limits each of the optimal match searches to taking at most time proportional to the length of the string.

5. Timesheet Part 2

Recall problem 4 from homework 7. Suppose we have N jobs labelled $1, ..., N$. For each job, you have determined the bonus of completing the job, $V_i \geq 0$, a penalty per day that you accumulate for not doing the job, $P_i 0$, and the days required for you to successfully complete the job $R_i > 0$.

Now, what we did not tell you last time is that we have a time limit of T days, in which we can choose to work on some of these jobs in only those T days. Given this information, what is the optimal job scheduling policy with a time limit of T days? Notice that 0 is a lower bound since we can choose to do no jobs at all if all of them happen to have negative value, or all of them take more than time T.

Design an efficient dynamic programming algorithm that finds the optimal schedule.

As in hw 7, begin by ordering the jobs by decreasing P_i/R_i. One key difference between Timesheet parts 1 and 2, is that some jobs will not be completed, and some for days (especially towards the end) it makes sense to not start a job. For this problem there are exactly T subproblems, one for each of the time units. If a job is not completed by time T, then some penalty is still incurred, but only if that job was started (daysleft$< R_i$).

recurrence at some time T_i:

$$T_i = Max \begin{cases} V_i - t \times Pi + T_{i-1}, \text{daysleft}_i = 0, , & \text{complete a job } i \\ T_{i-1}, \text{daysleft}_i - 1, , & \text{work on a job } i \\ T_{i-1}, & \text{work on nothing} \end{cases}$$

As in hw7 once a job is started it is optimal for that job to be completed so that the reward collected, so there is no reason to work on any jobs nonconsecutively. So for each of the T subproblems there are N choices for which job to complete, plus one choice for not working on any job at all, and the time it takes to preprocess sort the jobs by P_i/R_i value, for a total runtime of $O(N(T + \log N))$. Although, the purpose of the sorting is to minimize the number of jobs that must be tested, so in reality each time-step T_i should not have to go through all N job possibilities. Then the running time for large T approximates to the running time in hw7: $O(N \log N)$.

Extra space for Problem 1

Continued from Page 1

In your search for these substrings, if you find a substring word $s_{1,j}$ that includes the starting character, then you know that a proper reconstruction can be made, and you can return.

Proof of correctness: At each step of the algorithm starting with the last character, a unique substring is tested where for any given j the algorithm will test all word possibilities from $s_{i,j}$ where $i \in \{1, \ldots, j\}$. Since the algorithm checks if all possible substrings are words (as necessary), if there exists a nonoverlapping set of words then the algorithm will find it.

We can discover whether each of these substrings is a word in $O(1)$ time, and there are at most n^2 dictionary checks for all of the possible substrings of s, so the total running time is $O(n^2)$

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

There are two way to go about this, you can either construct the upper triangular subsequence table as described above, or simply memoize each of the subsequence paths as you proceed through the recurrence. If you go the recurrence route, then at the moment you reach a substring word that includes the first character you can return all of the successive calls to that led you there, and your sequence of words for calls $j_0, \ldots, j_n$ will be $\{(1, j_n), (j_n + 1, j_{n-1}), \ldots, (j_1, j_0)\}$.

Extra space for Problem 2

Continued from Page 2

Proof of correctness: For each of the cut sites (i, j) you are maximizing the cost of the next cut as well as all of the cuts of the sub-box either $(i, k > j)$ or $(k > i, j)$, so by induction with the base case being $(i, j) = (X, Y)$ you know that the total selling price is maximized. If you find happen to find a resulting sub-optimal selling price then there must have been a case where the maximum was not chosen, which is contradictory to the recurrence above.

For each of the $X \times Y$ points you have to evaluate $4 * n$ different possible cuts. This makes for a total running time of $O(n \times X \times Y)$.