CS170–Fall 2012 — Solutions to Homework 6

Ben Augarten, section 106, `cs170-bo`

October 17, 2012

1. We want to find the minimum penalty to arrive at the last hotel. If we knew all of the minimum penalties for each hotel before the last, then we could very easily calculate the last one. The last one would be the minimum of $p(a_i) + (200 - (a_n - a_i))^2$, for $0 \leq i \leq n$. We can recursively apply this procedure (or do it iteratively to find the minimum penalty for each hotel). I.e. $p(a_0)$ (our starting point) is 0, $p(a_1) = (200 - a_1)^2$, $p(a_2) = \min((200 - a_1)^2 + (200 - (a_2 - a_1))^2, (200 - a_2)^2)$. So on and so forth. Also, keep previous pointers so we can get the sequence back.
   Correctness: For each a_i we know that we can calculate the minimum penalty using just the node behind the element. Therefore, we can compute the minimum penalty for each hotel in order, and we are guaranteed to get the right sequence that results in the minimum penalty for getting to a_n and we can follow the previous pointers back to get the complete sequence.
   Run Time: For each a_i we have to look at all i's before it. That is $1 + 2 + 3 + ... + n$ which is $O(n^2)$

2. (a) We keep a set of booleans, where $B[k]$ is true iff $s[k, .., n]$ can be split into words. We want to know if $B[1]$ is true or not. For any $B[i]$, we can determine it using ONLY $B[i, .., n]$ by looking for all words starting at $s[i]$. If a word, w, starts at $s[i]$ AND $B[i + \text{len}(w)]$ is true then $B[i]$ is true. We start the algorithm by setting $B[n]$ to true (the empty string is a valid word) and traversing backwards through the string. If we get to $s[1]$ and find that $B[1]$ is true then we can split the words, if $B[1]$ is false, then we can't split the words.
   Correctness: For each $s[i]$, we can split all the words after that i iff there is a word, that ends where we can already split the string – but we are keeping track of where we can split the string, so we can just check $B[i + \text{len}(word)]$ and if its true then that is a valid split. We can keep going until we get to index 1 and check if there is a valid split at 1. If so, then we know we can split the string.
   Run time: For each i we only look at at most $n - i$ elements, checking them in the dictionary to see if they are words. That means we look at at most $1 + 2 + 3 + .. + n$ letters throughout the algorithm. Because dictionary lookups are constant time, this leads to a total running time of $O(n^2)$

   (b) To keep the sequence of words, for each $B[i]$ set it to $(true, word)$ where $word$ is the word starting at $s[i]$ that makes the split at i valid. Then we can reconstruct the list by looking at $B[1].word$, $B[1 + len(word)].word$, etc.

3. Given an X by Y rectangular piece of cloth, we have to find the maximum value we can get from cutting it. Obviously the maximum profit we can get from a piece of cloth is the maximum profit we can get from each distinct way we can cut it. Let P be a dict of profits, keyed on (x, y) where those are the dimensions of the rectangle. Similarly, let $V(x, y)$ be the maximum immediate value of a particular rectangle. That is, $V(x, y) = 0$ if there is no c_i for this piece of cloth, or else its the maximum c_i for a piece of cloth with $(a_i, b_i) = (x, y)$. So, for an X by Y rectangle, its value is $\max(V(X, Y), P(1, Y) + P(X - 1, Y), P(2, Y) + P(X - 2, Y), ..., 2 * P(X/2, Y)), P(X, 1) + P(X, Y - 1), ..., 2 * P(X, Y/2))$. We can apply this definition recursively, getting the value of each and using symmetry to help speed up the algorithm a little bit (the value of X,Y is the same as the value of Y,X because we can change the orientation of each of the cuts).
   Correctness: If there is an optimal solution, this algorithm will find it. For each X,Y rectangle, the algorithm takes the maximum of the value of selling that, and any and all possible cuts you can make to the cloth. This is the maximum price you can get for that piece of cloth. The base case is when you get to the smallest possible sellable unit – at which $V(X, Y) > 0$ and any cut to it renders it worthless.
   Runtime: There are XY total rectangles that we can make out of the X by Y cloth. For each of these, you have to try at most $X + Y$ cuts and loop through n total products before being able to determine its maximum possible value. That leads to a run time of $O(XY(X + Y + n))$, which is quadratic in X, Y and linear in n

4. We can use the same approach as in class. How much do we have to alter the first string to get the second string. Aka, what is $E(n, m)$? Well....
   $E(n, m) = \min(\delta(-, y[m]) + E(n, m-1), \delta(x[n], -) + E(n-1, m), \delta(x[n], y[m]) + E(n-1, m-1))$. This is correct because for the first n chars of x and m of y, either the last characters are translated, or inline. If it requires an insertion, then we insert an $-$ to x and pop off the last y value, if it requires a deletion, then we add a $-$ to the y to match the last value of x and add the scores for both of these cases. Else, we can always take the score of the last element of y and x and then recursively get the score for the remainder of the list. Then we set the base cases: $E(i, 0) = i, E(0, j) = j$. Correctness follows from the above, the optimal solution has to fall into one of these three categories for every n, m.
   Runtime: Nested loops – for each $i \in [1, n]$ we have to run the algorithm $j \in [1, m]$ times, so its $O(nm)$

5. First, lets sort all of the triples according r_i, or where they end in ascending order. Now for each triple we can find out its maximum possible weight simply by getting the max of possible weight of all triples before it and adding the weight of the current triple if it doesn't overlap. That is, the first triple has maximum weight of itself. Then the next triple has maximum weight of the max of: the weight of the first, the weight of the second, or the weight of the first and the weight of the second IF the triples cover disjoint sets. We can do this for every triple, finding the optimal solution, which will just be the maximum of all of the maxes.
   Correctness: In the sorted version, the maximum weight we can get using intervals that do not go past the upper bound of a given interval can be computed using only intervals before it. Therefore, the maximum weight can just computed as shown above. We do this until we find the maximum possible weight for the last triple and then the maximum over all of these values produces the maximum possible weight that we can get. If we include previous pointers along with the maxes, then we can trace our way back. I.e. store the max as a tuple of (max, previous_index) where max is the maximum possible weight and previous_index is the index in the sorted list of the interval that we used in calculating the max. Of course, some of the maxes of the intervals won't actually contain the interval itself. Therefore, for every triple, we set the previous_index to the first triple in the list that has the maximum value if there are duplicates. Likewise, when traversing the list after assigning values to every element, we accept the first maximum element if there are duplicates, to avoid including triples that got their max value from a previous element that overlapped with its range. This will produce a correct set of disjoint sets because it excludes all sets that overlap with the optimal solution.
   Runtime: the runtime is $O(N^2)$ where N is the number of triples given. It takes $O(N \log(N))$ to sort them, then for each one you have to check at most N other maximums before assigning it a value, so its $O(N \log(N) + N^2) = O(N^2)$