Problem Set 5

All parts are due Thursday, December 8, 2016 at 11:59PM.

Name: Dimitris Koutentakis

Collaborators: Nestoras Hahamis, Carl Unger, Driss Hafdi

Part A

Problem 5-1.

(a) In order to compute the optimal alignment score, we should use an algorithm similar to that of part B, without the parent pointers. In specific, we have to first make a grid (list of lists) of size $O(n^2)$ where each cell is used to calculate the score of the specific sequence. The grid is indexed by letters A-G depending on the sequence we're evaluating. Once we have the $O(n^2)$ grid, we have to initialize the first column and row by starting with 0 and adding 4 to each cell next/below it, representing a series of "_ " matching to a letter.Now that we have our grid fully initialized, we want to start with a double loop calculating possible values at each step. At each step we calculate the value of each possible option added to the according neighboring cell. For example, for each cell we check which of: {cell on top plus 4, cell on the left plus 4, diagonal cell plus according score based on matrix} is smaller. Then we update the cell with that score. The value in the in the lowest-rightmost cell will be the score of the optimal alignment.

(b) In order to lower the space complexity from $O(n^2)$ to $O(n)$, we have to modify our algorithm.

What we need to do is as simple as storing two rows of the matrix of (a) at every step. Since we never use more cells than the row above, we can easily ignore all rows further than that. Thus the algorithm would be exactly the same, but instead of initializing and using an array of size $O(n^2)$, we will be using an array of size $O(n)$.

(c) In order to keep track of the optimal alignment sequence, we have to use the technique used for the coding problem in part B. Namely, we have to use parent pointers, so that we can keep track of the optimal sequence without needing to use any more space.

The way we will solve the problem is to still have an array of size $O(n^2)$, but in every cell, instead of just storing the score of the optimal alignment sequence, we will also be storing the current value and a vector pointing to the parent. In every step, when calculating the optimal score, we select the parent cell and the new value that have the minimum score. Then we store in that cell a tuple that has: (current best move, (x.coord,y.coord), score). We always base our selection of letter on the 3rd entry.

In order to return the sequence, we trace the parent vectors, starting from the lowest rightmost cell. From there, we follow the second entry to the parent with the lowest score (stored when calculating the new lowest store) and we add the letter corresponding to that cell to our sequence. We keep doing that until we reach the uppermost, leftmost cell. Then we return the sequence traced.

(d) In order to modify the algorithm for (b) to run in $O(n)$ space, we need to implement a combination of Divide and Conquer with Dynamic Programming.

The overall optimal sequence for two strings will simply be a merge of optimal the optimal sequences of substrings of the original strings, following the optimal substructure principle. This is what allows us to implement Divide and Conquer in this problem. Thus we can solve this problem just by splitting each string in half, so we have $a_1 = a[0, \lfloor n/2 \rfloor]$ and $a_2 = a[\lfloor n/2 \rfloor + 1, n-1]$ and reverse the list a_2. Then, in order to find where to split list b (on the vertical axis of the imaginary grid), we have to have find which point will result in the minimum score. We can do that in O(n) space using the algorithm in part (c). Once we find that point k, we recurse the problem on the sublists of $a_1 = a[0, \lfloor n/2 \rfloor]$ and $a_2 = a[\lfloor n/2 \rfloor + 1, n-1]$and $b_1 = [0, k]$ and $b_2 = [k, 0]$.

This algorithm obviously has time complexity $O(n^2)$ (since we have to do $O(n^2)$ operations) and has space complexity of $O(n)$, since we never store the entire nXn grid.

(e) In order to modify the above algorithm to run with a maximum of 100 gaps, we need to add another dimension to our DP algorithm. Now, every subproblem will be a function of i (string a), j (string b) and k (the number of gaps), so we solve a $DP(i, j, k)$. We also impose the constraint of k¡=100 (less than 100 gaps). This in turn implies that the difference between i and j cannot be more than 100, since in the worst case scenario adding 100 gaps to one string will mean that abs(i-j)¡=100.

The running time seems to be $O(n^3)$, but is in fact $O(n^2)$. This can be seen from the fact that k is a constant and i and j can only differ up to a constant amount of time. Thus this constrained 3D DP algorithm will run in $O(n^2)$.

Problem 5-2.

(a) The algorithm provided by Prof. Caufield will not always work. for example, consider the following city price list: p=[2,1,2,10]. His algorithm will start with city No. 2 (cost 1) and that will cover cities 1 and 3 (each cost 2). Then he will have to run advertising on the last city for a cost of 10 resulting in a total cost of 11. The optimal would be a cost of 4, by advertising in the first and third cities.

(b) A better approach to solving this problem would be a Dynamic Programming algorithm. In specific, we would have to create a one dimensional dp array, e.g. $dp[p_0]$. The iterations for the solution of this problem would be n. In each iteration we would have to consider whether it's cheaper to advertize at that city or the one before it in order to get it covered. That is at each iteration we need: $DP[i] = min(DP[i-2] + p_i, DP[i-3] + p_{i-1})$. The final result will be DP[n-1].

Since each step takes constant time, the time complexity of this algorithm will be O(n).

(c) In order to modify the algorithm for general k, we should run the following version of the above Dynamic Programming algorithm.

We begin by initializing a one-dimensional DP array to DP[0] = p_0. We can easily determine that the number of sub-problems to solve is n, since we need to solve for DP[0] up to DP[n-1]. On each step of the algorithm, we will do: $DP[i] = \min_{j\in[0,k]}(DP[i-j-k] + p_{i-j})$. This checks if it is cheaper to advertise directly at c_i or at the previous city c_{i-1} in order to have cities up to c_i covered. The solution to our problem will then be DP[n-1]

The time complexity of this algorithm is O(nk), since the complexity of solving each sub-problems is O(k).

Part B

Problem 5-3.

(a) *Submitted on alg.csail.mit.edu*

(b) *Submitted on alg.csail.mit.edu*

(c) In order to use a graph to solve this problem we would first have to construct a graph in the form of an array of size: $l_1 \times l_2 \times l_3$. In this graph, the node G[0][0][0] is the state where all ghosts are alive, but no killing moves have happened. At G[i][0][0], ghost2 and ghost 3 still have all their lives, but ghost 1 has lost a total number of i lives. Same goes for the other two dimensions of the array. Therefore, all three ghosts will be dead at the state G[l_1][l_2][l_3]. Each edge from one vertex to another represents a loss of life. That is from G[i][j][k] either Ghost1 loses a life (G[i + 1][j][k]), Ghost 2 loses a life (G[i][j + 1][k]), Ghost 3 loses a life (G[i][j][k + 1]), or any combination of the above.

Since the graph is Directed and Acyclic, our problem will be the same as finding the shortest path from G[0][0][0] to G[l_1][l_2][l_3]. This is done using Dynamic programming as in part b, where DP[i][j][k] is an array of the size of the graph, and contains another array containing [shortest path from (0,0,0) to (i,j,k), parent pointer], and DP[0][0][0] = [0, (0,0,0)].

(d) Let's consider the problem as a graph described in (c). The only difference from (c) is that this graph will have i dimensions, one for every ghost. Each edge is a combination from moving exactly one index away from a given position in any dimension. It is important to realize that not all of the edges will exist for any vertex, but they will depend on if a move to remove a life from a given set of ghosts is the same. In this scenario diagonal edges between different dimensions are possible, and must be checked by our algorithm.

(e) In order to improve our algorithm to include this symbol, we just have to include one more case for checking. In every step/iteration of our algorithm, if one of the moves to remove a life for a given ghost is the extra character given, we will add the possibility of all ghosts losing a life (regardless of the character needed to kill them), and recurse on DP-ghost(ghost1, ghost2, ghost3, i - 1, j - 1, k - 1). This is the same as if the move to remove a life was the same for all ghosts.