CSE 202 - Homework 2

1.1 Maximum Length Chain of Subwords

High level description:
To find a maximum length chain of subwords in a given list of words, we first sort them by their string lengths. Since a subword of a word is always of lower length than the word, the order of subword chains is preserved in this sorted list. We then compute the maximum length of chain till a particular string. This is done by checking if any of the preceding strings are potential subwords and hence the maximum chain till such potential strings. Since, there arise overlapping sub-problems where we might have to check the maximum length of chain till a certain string multiple times, we use dynamic programming to solve this problem.

Algorithm 1 (Using Dynamic Programming and KMP Algorithm):

- Sort the set of string in a non-decreasing order of their string lengths. Let S = [s_1,s_2,....s_n] be the list of strings after sorting.
- **Dynamic Programming Definition:**
  For each s_i in S, we compute the maximum length of chain of subwords till s_i. Suppose $MaxLCS(s_i)$ denote such maximum length of chain till s_i for $1 \leq i \leq n$. The overall maximum length of chain is given by $max(MaxLCS(s_i))$ for $1 \leq i \leq n$. We also keep track of the chain through $previous(i)$ at a index i ($1 \leq i \leq n$) which denotes the index of the previous/last but one element in the maximum length chain of subwords ending at s_i.
- **Recursive Formulation:**
  To compute $MaxLCS(s_i)$, we compare s_i with each of the string before index i i.e. s_1 through s_{i-1}. We check if each string s_j where $1 \leq j \leq (i-1)$ can be a subword of s_i. We do this through a pattern searching algorithm i.e. Knuth-Morris-Pratt algorithm where we search for s_j in s_i.

  Suppose we find q such valid subwords s_{i_p} of s_i where $1 \leq p \leq q$, then:

  $$MaxLCS(s_i) = 1 + max(MaxLCS(s_{i_p}))$$
  $$\text{where } i > i_p \text{ and } 1 \leq p \leq q$$

  The $previous(i)$ is then set to such value $i_{p'}$ where the maximum of $MaxLCS(s_{i_p})$ occurs i.e. $previous(i) = i_{p'}$ where $MaxLCS(s_{i_{p'}}) = max(MaxLCS(s_{i_p}))$ and $1 \leq p^{'} \leq q$.

  If no such valid subwords are found for s_i, then $MaxLCS(s_i) = 1$ and $previous(i) = i$ i.e the longest chain ending at index i is the one starting at index i.

  Base Cases : At the base case where $i = 1$, $MaxLCS(s_1) = 1$ and $previous(1) = 1$ since the longest chain starts and ends at s_1.
- **Algorithm:**
  To find the maximum length chain of subwords in S = [s_1,s_2,....s_n], we thus invoke $MaxLCS(s_n)$.

We keep track of the maximum of all such $MaxLCS(s_i)$ computed at the recursive calls of the function and index where the maximum is encountered. This maximum occurring at index $maxChainIndex$ is the overall maximum length of the subword chain. To trace back the maximum length chain, we use $previous(i)$. Starting from the index $maxChainIndex$, we add the string at that index in S to a list L and update the $maxChainIndex$ with the value in $previous(maxChainIndex)$. We repeat and append such strings to list L until we encounter $maxChainIndex = previous(maxChainIndex)$ i.e. the string is pointing to itself. The reversed list L is the required output of the form w_1,w_2,....w_t where $w_{i+1} = uw_iv$.

Complexity of Algorithm 1:
For sorting strings in S, the time complexity is $O(nlogn)$. For each string s_i in S, we perform a pattern matching with each of the preceding strings. The number of such pattern matches is therefore of the order of n^2 where n is the number of strings in S. For each pattern match using the Knuth-Morris-Pratt algorithm, the time complexity is of the order $O(k)$ where k is the maximum possible length of string in S. Hence the overall time complexity of the algorithm is $O(n^2k)$.

Correctness of Algorithm 1:
In the base case where $i = 1$, the longest chain of subwords starts and ends at index 1 as no more elements exist to the left of s_1 in the sorted list. Therefore $MaxLCS(s_1) = 1$. For all other s_i, the value of $MaxLCS(s_i)$ is either obtained as 1 (in cases where pattern matching with all previous strings in S does not result in a successful match) or $1 + MaxLCS(s_j)$ where $1 \leq j < i$ and $MaxLCS(s_j)$ is the maximum length of subword chain till index j (through induction). Since, we iterate through all strings before s_i in the computation of $MaxLCS(s_i)$ and choose j such that $MaxLCS(s_j)$ = max($MaxLCS(s_1)$,$MaxLCS(s_2)$,... ,$MaxLCS(s_{i-1})$), the value of $MaxLCS(s_i)$ is maximized for the given list of strings.

Algorithm 2 (Using an Ordered Map Data Structure):

- Sort the set of string in a non-decreasing order of their string lengths. Let S = [s_1,s_2,....s_n] be the list of strings after sorting.
- Initialize an empty ordered map data structure M. The map is used for storing key values pairs $(s_i, MaxLCS(s_i))$ where $MaxLCS(s_i)$ denotes the maximum length chain of subwords till index i. We also initialize an array named $previous$ of size n. The array element at a index ind denotes the index of the previous/last but one element in the maximum length chain of subwords ending at s_{ind}.
- For each string s_i in S, we construct all the possible subwords of s_i. This can be done using two pointers $start$ and end, initially pointing to the first letter and the last letter of s_i respectively. Keeping the $start$ pointer fixed, we move the end pointer to the left until $end \geq start$. We then move the $start$ pointer one position to the right, end pointer to the last letter and repeat the process. At each iteration the subword is the string from the $start$ pointer to the end pointer both inclusive.
- For all the subwords of s_i, we check if a pair with a key s_i exists in the map data structure M. If none of the subwords are present as a key in M, we insert the key value pair $(s_i, 1)$ into the map. Update the array element $previous[i]$ to i since the longest chain including s_i starts and ends at i.Else if we find in the map k key value pairs $(s_{i_p}, MaxLCS(s_{i_p}))$ corresponding to subwords s_{i_p} of s_i, we insert $(s_i, maxPrevChainLen + 1)$ into M where $maxPrevChainLen = max(MaxLCS(s_{i_p}))$ for all

$1 \leq p \leq k$. Update the array element $previous[i]$ to such $i_{p'}$ where the maximum previous chain is found i.e. $MaxLCS(s_{i_{p'}}) = maxPrevChainLen$.

- The overall maximum length chain of subwords for S is the maximum of all values $MaxLCS(s_i)$ inserted into the map M for string s_i in S where $1 \leq i \leq n$. Therefore while inserting key value pairs into the string we keep a track of the maximum value inserted so far and the index $maxChainIndex$ of the string where the maximum values occurred.
- To trace back the maximum length chain, we use the array $previous$. Starting from the index $maxChainIndex$, we add the string at that index in S to a list L and update the $maxChainIndex$ with the value in $previous[maxChainIndex]$. We repeat and append such strings to list L until we encounter $maxChainIndex = previous[maxChainIndex]$ i.e. the string is pointing to itself. The reversed list L is the required output of the form w_1,w_2,....w_t where $w_{i+1} = uw_iv$.

Complexity of Algorithm 2:
For sorting strings in S, the time complexity is $O(nlogn)$. For each s_i in S, we construct the subwords using two pointers. Since both these pointers iterate through the string whose maximum possible length is k, the complexity of this step for each word is of the order k^2. For each subword, we search and insert in an ordered map. Worst case time of both these operation is $O(logn)$. Since there are n strings in the list for which all these operations are performed, the overall time complexity is $O(k^2nlogn)$.

Correctness of Algorithm 2:
In the base case where the first element is processed, the longest chain of subwords starts and ends at index 1 as no more elements exist to the left of s_1 in the sorted list. Therefore $MaxLCS(s_1) = 1$ and the key value pair (s_1,1) is put into the map. For all other keys s_i, the value $MaxLCS(s_i)$ is either obtained as 1 (in cases where its subwords do not exist in the map) or $1 + MaxLCS(s_j)$ where $1 \leq j < i$, s_j is a subword of s_i and $MaxLCS(s_j)$ is the value for the key s_j in the map or in other words the maximum length of subword chain till index j (through induction). Since, we construct all subwords of s_i in the computation of $MaxLCS(s_i)$ and check for the maximum of maximum length chains ending at these subwords if they exist in the map (i.e. they exist before index i in S), the value for the key s_i i.e $MaxLCS(s_i)$ is maximized for the given list of strings.

1.2 Classes and Rooms

High level description:
The strategy while assigning the classrooms to classes is to apply the greedy method and assign the least possible capacity room satisfying the enrollment constraint to each class sorted by the enrollment sizes to minimize the total size of the rooms assigned.

Algorithm:

- Sort the list of classes C in a non-decreasing order of their enrollment. Let the number of classes be n and C_s = [c_1,c_2,....c_n] be the list of classes after sorting.
- Sort the list of classrooms R in a non-decreasing order of their capacity. Let the number of classrooms be m and R_s = [r_1,r_2,....r_m] be the list of classrooms after sorting.

- Let $A_g(c_i)$ denote the room assigned by this greedy algorithm for the class c_i in C_s. For each c_i in C_s starting with $i = 1$, we iterate through R_s to find a minimum value of j such that enrollment of c_i is less than or equal to the capacity of r_j i.e. $E(c_i) \leq S(r_j)$. If there exists a valid assignment r_j for c_i, then $A_g(c_i) = r_j$ .While processing the subsequent class i.e. c_{i+1} (where $c_{i+1} \in C_s$) we only iterate in the list of rooms starting from r_{j+1}.
- Cases of no valid assignment: If the number of classrooms is less than the number of classes i.e $m < n$, there exists no possible valid assignment. Also, if there exists a c_i in C_s such that it enrollment is greater than the maximum possible capacity available for the rooms i.e $E(c_i) > S(r_m)$, a valid assignment does not exist.

Complexity:
Time complexities for sorting the list of classes and classrooms are $O(nlogn)$ and $O(mlogm)$ respectively. In the case where a valid assignment exists for each c_i in C_s, we traverse the complete list C_s and in turn for each c_i, we traverse a finite number of elements m_i from R_s. In the worst case i.e when all the rooms are checked for a valid assignment, $\Sigma m_i = m$. Therefore the entire traversal for the assignment takes O(n+m) time. Overall time complexity of the algorithm is therefore $O(mlogm)$ since $m \geq n$ for valid assignment.

Correctness:
Claim: The greedy algorithm outlined above minimizes the total sizes of the rooms used in cases where a valid assignment exists. $A_g(c_i)$ denotes the room assigned by the greedy algorithm for the class c_i in C_s. Let T_g denote the total sizes of all classrooms assigned. Then,

$$T_g = \Sigma_{i=1}^{n} S(A_g(c_i))$$

To prove our claim by contradiction, we assume there exists an optimal algorithm which assigns a net capacity of less than T_g for given classes and classrooms. Let $A_o(c_i)$ denote the room assigned by the optimal algorithm for the class c_i in C_s and T_o denote the total sizes of all classrooms assigned. Then,

$$T_o = \Sigma_{i=1}^{n} S(A_o(c_i)) \leq T_g$$

Consider the list of classes, classroom pairs assigned by each of the greedy and optimal algorithms. Sort these lists by the non decreasing order of enrollment of the classes. The lists are then of the form L_g = $[(c_1, A_g(c_1)), (c_2, A_g(c_2)), ..., (c_n, A_g(c_n))]$ (assignment by greedy algorithm) and L_o = $[(c_1, A_o(c_1)), (c_2$ $, A_o(c_2)), ..., (c_n, A_o(c_n))]$ (assignment by optimal algorithm). Since the net capacity assigned by both these algorithms is different, there must exist at least one class where the assignment differs. Let c_i be the first class where this discrepancy is observed i.e if $A_g(c_i) = r_j$ and $A_o(c_i) = r_{j'}$ then $r_j \neq r_{j'}$. This condition can only arise in the following two cases.

Case - 1: The room r_j is not used in the assignment by the optimal algorithm. Since greedy algorithm picks the classroom that has minimum capacity while meeting the enrollment criteria for the class, $r_j \leq r_{j'}$. Therefore, $r_{j'}$ can be swapped with r_j in the assignments by optimal algorithm.

Case - 2: The room r_j is assigned to a class c_k where $k \neq i$ by the optimal algorithm. The pairs $(c_i, r_{j'})$ and (c_k, r_j) exist in assignments by optimal algorithm where $k > i$ (first discrepancy occurs at index i). Since $k > i$ and (c_i, r_j), (c_k, r_j) are valid assignments for c_i in greedy and optimal solutions,

$E(c_i) \leq E(c_k) \leq S(r_j)$. Also since greedy algorithm picks the classroom that has minimum capacity while meeting the enrollment criteria for the class c_i, $r_j \leq r_{j_\prime}$. Therefore, the rooms assigned for the pairs $(c_i, r_{j\prime})$ and (c_k, r_j) by optimal solution can be swapped.

By exchange argument in the above two cases that arise, the optimal solution can be modified to make similar assignments as the greedy without increasing the total capacity assigned. Hence, there exists no better solution than that is obtained by the greedy algorithm outlined above to minimize the total room sizes.

1.3 Business Plan

High level description:
The strategy to select the projects such that the capital is maximized before the IPO is to apply greedy method and pick the project with the maximum profit at any step within the given constraint of the cumulative capital available so far.

Algorithm:

- Sort the projects in a non-decreasing order of their capitals. To break ties while ordering c_i values, order such pairs by non-increasing values of p_i. Let [i_1,i_2,....i_n] be the list of projects after sorting. Their corresponding capitals c_{i_j} and profits p_{i_j} are (c_{i_1},p_{i_1}),(c_{i_2},p_{i_2}),....,(c_{i_n},p_{i_n}). Initialize an empty max heap MH to store the pairs of eligible projects and their profits. The pairs in the heap are ordered by the profit component.
- To select $(j+1)^{th}$ project i_{j+1} where $0 \leq j < k$ and the accumulated capital is C_j, iterate through the sorted list of available projects until the capitals of the project are less than or equal to C_j, remove them from the list of available projects and push these projects along with their corresponding profits into the max heap MH.
- We then pop the top of the max heap to get a project i_k with profit p_{i_k}, and add it to the list of selected projects L_g (initially empty). The accumulated capital C_{j+1} is the accumulated capital till the first j projects along with the profit on the $(j+1)^{th}$ project i.e. $C_{j+1} = C_j + p_{i_k}$.
- If the max heap MH is empty or we have selected k projects, we output the list L_g of selected projects and the capital accumulated $C_{k'}$ where $k^{'} \leq k$.

Complexity:
The time complexity for sorting the projects is $O(nlogn)$. Iterating over the list of eligible projects take a time of the order $O(n)$ and putting the projects into the max heap takes a worst case time complexity of $O(nlogn)$ when there are n projects in the heap. Therefore the overall time complexity is $O(nlogn)$

Correctness:
Claim : The greedy algorithm outlined above maximizes the total profit that can be earned before the IPO. Let the list of the projects selected by the greedy algorithm be L_g = $[i_{g_1}, i_{g_2}, ..., i_{g_{k'}}]$ where $k^{'} \leq k$. For the base case i.e while selecting the first project, the greedy algorithm selects the project with the maximum profit from a list of projects whose capital required is less than or equal to C_0.Let C_g denote the cumulative profit before the IPO. Then,

$$C_g = \Sigma_{l=1}^{k'} p_{i_{g_l}}$$

To prove our claim by contradiction, we assume there exists an optimal algorithm which results in a higher profit than C_g. Let $L_o = [i_{o_1}, i_{o_2}, ..., i_{o_{k''}}]$ be the projects picked by the optimal algorithm and C_o cumulative profit before the IPO by this method. Then,

$$C_o = \Sigma_{l=1}^{k''} p_{i_{o_l}} \text{ and } C_o > C_g$$

Since the projects picked by both these algorithms are different, there must exist at least one iteration where a different project is picked. Let us assume that while picking the j^{th} project, the greedy algorithm chose the project i_{g_j} whereas optimal solution chose i_{o_j} where $g_j \neq o_j$. We know that the greedy method chooses the project with the greatest profit within the constraint of accumulated capital till that point i.e. $p_{i_{g_j}} > p_{i_{o_j}}$ and hence accumulated capital for greedy solution is more than the accumulated capital for the optimal solution after choosing the j^{th} project. Since all the projects post the j^{th} step can be accomplished with the capital by picking i_{o_j}, they can also be accomplished by picking i_{g_j}. In the optimal solution, we can swap i_{o_j} with i_{g_j} at the j^{th} step. By this exchange argument and by induction in further steps, the optimal solution can essentially made equivalent to the greedy solution without decreasing C_o. Therefore $C_o = C_g$ and hence there can exist no solution with a capital accumulated before IPO more than that obtained by the greedy solution.

1.4 Speech Recognition

High level description:
To find if a given sequence $\sigma_1, \sigma_2, ...\sigma_k$ exists in a directed graph G, starting from a vertex v_0, we solve this recursively by arriving at a node v_j up to which the prefix $\sigma_1, \sigma_2, ...\sigma_i$ is found and we further intend to find the suffix $\sigma_{i+1}, ...\sigma_k$. Since, there exist overlapping sub-problems where we may arrive at the same vertex looking for the same suffix, we use dynamic programming to solve this.

Dynamic Programming Definition:
Suppose in the process of finding the required path $\sigma_1, \sigma_2, ...\sigma_k$, we are now at a vertex v_j and the suffix to be traced from that vertex is σ_i through σ_k where $1 \leq i \leq k$. We define $path$(node = v_j, suffixIndexStart = i) which checks if the given suffix $[\sigma_i, \sigma_{i+1}, ...\sigma_k]$ exists in the graph starting from the given node. The overall path in G is thus obtained by invoking $path$(node = v_0, suffixIndexStart = 1). We also define $tracePath$(node = v_j, suffixIndexStart = i) which maintains a list of paths(list of vertices) originating from v_j and labelled $[\sigma_i, \sigma_{i+1}, ...\sigma_k]$. $tracePath$ is updated only when the $path$ from that node and given suffix returns true values from the base case.

Recursive formulation:
Starting from a node v_j, if we have to trace the suffix $[\sigma_i, \sigma_{i+1}, ...\sigma_k]$ in the graph G, we first check for all the edge labels of edges connected to v_j. To be able to find the suffix from that node, there must be at least one edge with the label σ_i. Suppose there exist m nodes v_{j_1}, v_{j_2},...v_{j_m} connected to v_j such that the edges between v_j and these nodes is labelled σ_i. We the try to find the next suffix i.e. σ_{i+1} through σ_k from each of these m nodes. If at least one of further m recursive calls returns that the path exists, the required path exists in the graph G. The recursive formulation is as follows:

$path$(node = v_j, suffixIndexStart = i) = any($path$(node = v_{j_p}, suffixIndexStart = $i+1$))
where $1 \leq p \leq m$ and $\sigma(v_j, v_{j_p}) = \sigma_i$
(the function $any()$ returns true if at least one of its arguments is true)

$tracePath$(node = v_j, suffixIndexStart = i) is updated as a list of all such lists $tracePath$(node = v_{j_p}, suffixIndexStart = $i+1$) appended with the node v_j. (where $1 \leq p \leq m$, $\sigma(v_j, v_{j_p}) = \sigma_i$ and $path$(node = v_{j_p}, suffixIndexStart = $i+1$) is true)

Bases Cases: At any vertex v_j if the suffix has been fully traced, we output that the path exists in the graph G i.e. $path$(node = v_j, suffixIndexStart = $k+1$) outputs that path exists. Else while computing $path$(node = v_j, suffixIndexStart = i), if no vertex v_{j_p} connected to v_j exists such that $\sigma(v_j, v_{j_p}) = \sigma_i$ we output that the required suffix does not exist from that node i.e. No_Such_Path.

Complexity:
In the worst case in a densely connected graph, we check each vertex with every other vertex in an adjacency matrix for an edge between them that can be a part of sequence (one of σ_1 through σ_k). Therefore the overall time complexity is $O(n^2k)$.

Correctness:
In the base case where suffixIndexStart = $k+1$, it is returned that the path exists. This is true as the recursive call contains this parameter only after the pattern $\sigma_1, \sigma_2, ...\sigma_k$ has been realized. In all other cases, $path$(node = v_j, suffixIndexStart = i) is called where $1 \leq i \leq k$ which is called only if σ_1 through σ_{i-1} is found till vertex v_j and returns true only if it reaches the base case suffixIndexStart = $k+1$ which occurs when a path labelled σ_i through σ_k originates from v_j. In all other cases, the recursive formulation returns No_Such_Path. Also the path is traced only when all $k+1$ recursive calls arising from v_0 and terminating at the base case return true. Therefore, the algorithm holds good for any given directed graph G and sequence s.
