CSE 202: Take-Home Final

Jason Ramirez

December 8, 2017

Problem 1: Sequence Matching

Let T and P be respectively two sequences $t_1, \ldots, t_n$ and $p_1, \ldots, p_k$ of characters such that $k \leq n$. Let $v(i)$ denote the value for the i-th position in T for $1 \leq i \leq n$. The algorithm begins by creating a list $MaxSS$ that keeps track of the values and ordered list of character indices, (v_i, lss_i), of the maximum value subsequences of P found in T. All v_i and lss_i are initialized to $-\infty$ and an empty list [], respectively. The number of elements in $MaxSS$ is $k + (k-1) + (k-2) + \cdots + 1 = \frac{(k+1)(k)}{2}$. The pattern subsequences that the elements of $MaxSS$ correpond to, beginning with the lowest index of $MaxSS$, are: $p_1, p_1p_2, \ldots, p_1p_2 \ldots p_k, p_2, p_2p_3, p_2p_3 \ldots p_k, \ldots, p_{k-1}p_k, p_k$. Let S be the set of indices of $MaxSS$ that correspond to pattern subsequences of length 1, i.e.
$S = \{1, k+1, k+(k-1)+1, \ldots, k+(k-1)+\cdots+1)\}$.

A hashmap PI is created with a key for each distinct element p_i that is mapped to an ordered list of the indices lp_i of $MaxSS$ such that p_i is the last character in the subsequences indexed by those indices. More specifically, when p_i is found when traversing P, if p_i is not in PI then add key-value pair
$(p_i, [i, k+(i-1), k+k-1+(i-2), \ldots, k+k-1+\cdots+2+(i-(k-1))])$, while $(i-l) \geq 1$. So for a new p_4 the key-value pair would be
$(p_4, [4, k+3, k+k-1+2, k+k-1+k-2+1])$ If p_i is already in PI then just append the aforementioned list of indices to the list already in the hashmap.

The algorithm continues as follows:

1. It iterates through T, checking if each t_i encountered is a key of PI
   (a) If not, the algorithm continues
   (b) If so, it retrieves the list lp_j that t_i is the key for. $lp_j = [i_1, i_2, \ldots, i_m]$ is then iterated through and each index i_j is used to update each (v_{i_j}, lss_{i_j})
      i. A copy of $MaxSS$ is made, $MaxSSPrev$ is set equal to $MaxSS$ and $MaxSS$ is set equal to the copy.
      ii. It first checks if l_j is in S, and if so checks if $v_i > MaxSSPrev.v_{i_j}$ and if so setting $MaxSS.v_{i_j} = v_i$ and $MaxSS.lss_{i_j} = [i]$, otherwise if $v_i + MaxSSPrev.v_{i_j-1} > MaxSSPrev.v_{i_j}$ set $MaxSS.v_{i_j} = v_i$ and set $MaxSS.lss_{i_j} = MaxSSPrev.lss_{i_j-1}.append(i)$.

2. The algorithm returns $MaxSS[k]$, the value and ordered indices of the subsequence of T that matches the pattern P with maximum value.

Proof of Correctness:

Claim: The algorithm finds the sequence of indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that for all $1 \leq j \leq k$, we have $t_{i_j} = p_j$ and $\sum_{k=1}^{j} v(i_j)$ is maximized.

Proof. Clearly, after PI is created and initialized each distinct p_i is paired with the list of indices l_{p_i} such that the subsequences in $MaxSS$ located at these indices end with p_i. Consider the first iteration of the part of the algorithm that iterates through T. The

algorithm checks if $t_1 \in P$, if not, the algorithm continues since this t_1 cannot be part of any of the subsequences in $MaxSS$. If $t_1 \in P$ then $t_1 = p_j$ for some j and the indices in the list $l_{p_j} = [i_1, i_2, \ldots, i_l]$ update the corresponding values and lists (if needed) in the new $MaxSS$ by comparisons with $MaxSSPrev$. Although some pattern subsequences of length $m \geq 2$ may end with t_1, the values of the corresponding subsequences of length $m-1$ of these subsequences $p_i p_{i+1} \ldots t_1$ will have value $-\infty$ in $prevMaxSS$ so they will essentially not be updated. Only $MaxSS[i_p].v = v(t_1) \neq -\infty$ at the end of this iteration, where $i_p \in S$ corresponds to a subsequence of length 1, as this update only involves comparing $MaxSSPrev[i_p].v = -\infty$ to $v(t_1) \geq 0$. Also, $MaxSS[i_p].lss = [t_1]$.

Assume that at iteration i $MaxSS$ contains the ordered sequence of indices of T of maximum sum possible so far, for each subsequence of P, that have been found up to and including t_i. t_{i+1} is the new element of T to be processed by algorithm. If $t_{i+1} \notin P$, the algorithm continues since t_i cannot be part of any subsequence of P. If $t_{i+1} \in P$, then $PI.t_{i+1} = l_{p_j} = [i_1, i_2, \ldots, i_l]$ where $p_j = t_{i+1}$.

For each $i_y \in p_j$, if $i_y \in S$ then $MaxSS[i_y]$ corresponds to t_{i+1}, which is updated if and only if $v(t_{i+1}) > MaxSSPrev[i_y].v$ by setting $MaxSS[i_y] = (v(t_{i+1}), [t_{i+1}])$. So $MaxSS[i_y]$ contains the index of maximum of the subsequence t_{i+1} found so far.

If $i_y \notin S$, then i_y corresponds to a subsequence of length $m \geq 2$ ending with t_{i+1}. $MaxSS[i_y]$ is updated if and only if $MaxSSPrev[i_y - 1].v + v(t_{i+1}) < MaxSSPrev[i_y]$, since an ordered set of indices of greater sum than the previous ordered set of indices that matches the subsequences $p_r p_{r+1} \ldots t_{i+1}$ can only be found at this iteration (if possible) by using the maximal sum and indices of the subsequences that ends right before t_{i+1} which is located at $MaxSSPrev[i_y - 1]$. If an update is necessary
$MaxSS[i_y] = (MaxSSPrev[i_y - 1].v + v(t_{i+1}), MaxSSPrev[i_y - 1].append(t_{i+1})$.

As $MaxSS[k]$ corresponds to the sequence $P = p_1 \ldots p_k$, $MaxSS[k]$ contains the sequence of indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that for all $1 \leq j \leq k$, we have $t_{i_j} = p_j$ and $\sum_{k=1}^{j} v(i_j)$ is maximized. □

Time Complexity:

It takes $O(k^2)$ time to initialize the values of $MaxSS$ and $O(k)$ time to iterate through P and $O(k)$ time to add (p_i, l_{p_i}) to PI or update the list hashed by p_i in PI, threby requiring

$O(k^2)$ time in total. Iterating through T, copying $MaxSS$, and updating the corresponding indices of t_i in the new $MaxSS$ takes $O(k^2)$ time. Therefore, the algorithm has an overall time complexity of $O(nk^2)$.

Problem 2: Covering points with gain

The algorithm first sorts the intervals in nondecreasing order by their finishing times. An array G is maintained using this order, where $G[i]$ corresponds to the total gain of the $i-th$ interval based on the points in $\{x_1, x_2, \ldots, x_n\}$ that intersect with it. This ordering will be used when referring to an interval $[s_i, f_i]$. Then, the starting times, finishing times, and x_i values are sorted in nondecreasing order in another array. If there are $s_i's$ and $x_i's$ or $f_i's$ and $x_i's$ with the same value, x_i will be to ordered before f_i and and after s_i. Whenever an s_i is encountered, all the x_j values found up f_i is are added to $G[i]$.

The algorithm proceeds using dynamic programming as follows:

- The value we want is $Opt(d) = (d_d, l_d)$, the maximal difference, $profit_d$, between point gains and intervals costs after the $d-th$ interval is considered, d_d, and the corresponding list of intervals l_d
- Base case: $Opt(1) = (0, [\emptyset])$ if $G[1] \leq 0$ and $Opt(1) = ([[s_1, f_1]], G[1])$ if $G[1] > 0$
- Suppose $i > 1$:
  - $f_i \geq f_j, \forall j < i$, therefore, $[s_i, f_i]$ is disjoint from all previous intervals if and only if $s_i > f_{i-1}$.
    - If $G[i] \leq 0 \Rightarrow Opt(i) = Opt(i-1)$
    - Otherwise, $Opt(i).d = Opt(i-1).d + G[i]$ and $Opt(i).l = Opt(i-1).l.append([s_i, f_i])$
  - If there exists an intersection between $[s_i, f_i]$ and some previous interval $[s_j, f_j] \in Opt(i-1).l$, then $f_j \geq s_i$. Consider such an interval. The overlapping component of intervals i and j is $[\max(s_i, s_j), f_j]$.
    - $[f_j, f_i]$ has weight z and $z > c_i$ then add $[s_i, f_i]$ to $Opt(i-1).l$ (since $Opt(i-1)$ is no longer going to be used, it is being altered, this doesn't represent the actual optimal values of $Opt(i-1)$ anymore)
      - If $s_j \geq s_i$, remove $[s_j, f_j]$ from $Opt(i-1).l$. Let z_2 be the weight of $[s_i, s_j]$ not already in $Opt(i-1).d$, then $Opt(i-1).d = Opt(i-1).d + z - c_i + z_2$.
      - Otherwise, $[s_j, s_i]$ has weight z_2 and if $z_2 \leq c_j$, remove $[s_j, f_j]$ from $Opt(i-1).l$. Then, $Opt(i-1).d = Opt(i-1).d + z$.
    - If $z \leq c_i$
      - If $s_j \geq s_i$, let z_2 be the not already considered weight of $[s_i, s_j]$ if $z_2 > c_i$ add $[s_i, f_i]$ to $Opt(i-1).l$, remove $[s_j, f_j]$ from $Opt(i-1).l$, and $Opt(i-1).d = Opt(i-1).d + z_2$. If $z_2 \leq c_i$ then compare $Gain[i]$ to $Gain[j]$ and update accordingly, only the one with greater benefit is kept.

· If $s_j \leq s_i$ then then compare $Gain[i]$ to $Gain[j]$ and update accordingly, only the one with greater benefit is kept.

* $Opt(i) = Opt(i-1)$

$Opt(i)$ is the output of the algorithm.

Time Complexity

The sorting process takes $O((n+d)log(n+d)$ time. Each iteration of the dynamic programming component takes $O((n+d)^2)$ time and there are $O(n)$ iterations. So the overall time complexity is $O(n(n+d)^2)$.

Problem 4: Cellular Network

Let $G = (V, E)$ be an undirected graph, where $V = \{s_1, s_2, \ldots, s_n\}$ is a finite set containing n sites and $e_{ij} \in E$ if and only if there is an edge between s_i and s_j. Each site s_i has a cost $c_i \geq 0$ of selecting it and each pair of selected sites s_i and s_j have a benefit $b_{ij} \geq 0$ if they have an edge between them. Clearly, vertices s_i that are not adjacent to any edges can be removed from V and edges e_{ij} with $b_{ij} = 0$ can be removed from E.

In order to select a subset $S' \subset V$ of sites that maximizes the difference ($profit$) between the sum of the edge benefits less the vertex costs, a flow network $H = (V_H, E_H, c)$, where c is a capacity function, is constructed as follows:

- $V_H = \{s, t\} \cup E \cup \{s_i | e_{ij} \in E\}$. Note: $e_{ij} = e_{ji}$ since G is undirected
- $E_H = \{s\} \times E \cup \{(e_{ij}, s_i), (e_{ij}, s_j) | e_{ij} \in E\} \cup \{s_i | e_{ij} \in E\} \times t$
  - Each $(s, e_{ij}) \in E_H$ has capacity b_{ij}
  - Each $(e_{ij}, s_i) \in E_H$ and $(e_{ij}, s_j) \in E_H$ has capacity ∞
  - Each $(s_i, t) \in E_H$ has capacity c_i

A minimum cut (A, B) of G_H is found using the Preflow-Push maximum flow algorithm. Then $S' = \{s_i | s_i \in A\}$ and we output S' as the set of sites that are to be selected for maximum $profit$.

Proof of Correctness:

There consists of a finite min-cut (A_f, B_f) consisting of the edges from s to e_{ij}.

Claim: $e_{ij} \in A$ if and only if $s_i, s_j \in A$. Therefore, the minimum cut (A, B) found by the algorithm contains a subset $V' \subset V$ and only the edges between pairs of vertex elements in A.

Proof. Suppose $e_{ij} \in A$ and WLOG $s_i \notin inA$, then since there exists an edge in G_H from e_{ij} to s_i with capacity ∞, so the capacity C of (A, B) is not minimal, a contradiction.

Suppose $e_i, e_j \in A$ but $e_{ij} \notin A$, assuming $e_{ij} \in E_H$. Then by adding e_{ij} to A and removing it from B, the C is decreased due to the positive capacity of the edge s, e_{ij} so C is not minimal, a contradiction. □

Claim: There exists a cut (A', B') representing the subset $V' \subset V$ and the edges between pairs of these vertices.

Proof. Clearly, in this cut, $A' = \{s\} \cup \{s_i | s_i \in V'\} \cup \{e_{ij} | s_i, s_j \in V'\}$ and $B' = V_H \setminus A'$. □

Claim: The subset S', corresponding to a min-cut $C = (A, B)$, that is output by the algorithm maxmimzes $profit$.

Proof. The summation Q of the capacities edges that cross C from A to B is as follows:

$$Q = \sum_{e_{ij} \notin A} b_{ij} + \sum_{s_i \in A} c_i$$

And since

$$\begin{aligned} profit &= \sum_{e_{ij} \in A} b_{ij} - \sum_{s_i \in A} c_i \\ &= \sum_{e_{ij} \in E} b_{ij} - \sum_{e_{ij} \notin A} b_{ij} - \sum_{s_i \in A} c_i \\ &= \sum_{e_{ij} \in E} b_{ij} - (\sum_{e_{ij} \notin A} b_{ij} + \sum_{s_i \in A} c_i) \\ &= \sum_{e_{ij} \in E} b_{ij} - Q \end{aligned}$$

then a maximization of $profit$ is equal to a minimization of Q. So the set S' corresponds to a subset of the sites that maximizes $profit$. □

Time Complexity

it takes $O(n^3)$ time to construct H. There are $O(n^2)$ edges and $O(n)$ vertices in G, so there are $O(n^2)$ nodes in H. Therefore, the time complexity of running the Preflow-Push maximum flow algorithm is $O(n^6)$.

Problem 5: Center selection with producers and consumers

Consider the set C^* of k producers that minimizes $\max_{q \in Q} d(q, C)$, and let this value be d^*.

Let $C' = \emptyset$, let $P' = P$, and let $Q' = V - P'$ For $\leq k$ iterations add the producer P_i such that P_i is closest to a point $q \in Q'$. Then remove all points in P' and Q' that are within a radius of $3d^*$ from P_i, from their respective sets. Add P_i to C' and continue with the next iteration if $Q' \neq \emptyset$, otherwise terminate the algorithm.

If $|C'| \leq k$ then clearly $\max_{q \in Q} d(q, C') \leq 3d^*$.

Claim: If $|C'| > k$ then then for any set of k producers C, $\max_{q \in Q} d(q, C) > d^*$.

Proof. Each $P_i \in C'$ is selected because it was the producer closest to any point $q_i \in Q'$. $d(q_i, P_j) \leq d^*$ for some $P_j \in C^*$, if $P_i = P_j$ then all $q_k \in Q'$ within a distance of $d*$ from P_i are removed from Q'. If $P_i \neq P_j$ then by triangle inequality $d(P_i, P_j) \leq 2d^*$ and again by the triangle inequality all points $q_k \in Q'$ within a distance of d^* from P_j are covered by P_i since $d(P_i, q_k) \leq 3d^*$. So if at any iteratAlso, since $d(q_{i+1}, P_j) > d^*$, $\forall P_j \in C^* \setminus P'$ where q_{i+1} is the next closest $q \in Q'$ to P', then it must correspond to a $P_k \in P'$. So every P_i chosen by the algorithm covers all consumers that are within a distance d^* of a distinct producer in C^*. So if more than $|C'| > k$ then $\max_{q \in Q} d(q, C^*) > d^*$. □

The algorithm is as follows: choose any $q \in Q'$ and add some P_i to C such that $\min_{P_i \in P} d(q, P_i)$. Remove q from Q' and add P to C and proceed through the algorithm as follows for at most $k-1$ iterations: choose $q \in Q'$ such that q is farthest from C add the closest $P_i \notin C$ to q to C and remove q from Q', if there are no $q \in Q'$, terminate the algorithm. C is returned.
Claim: The algorithm returns a set of producers of size k, C, such that $\max_{q \in Q} d(q, C) \leq 3d^*$.

Proof. Suppose $\max_{q \in Q} d(q, C) > 3d^*$, then there exists some $q \in Q$ that satisfies this condition. Since the second algorithm chooses a consumer $q \in Q'$ that is farthest from C_i, the cover at the beginning of iteration i of the algorithm, this implies that every q_i chosen at an iteration i of the algorithm is at least a distance $3d^*$ from C_i. But this is the procedure of the first algorithm, so by choice of $d*$ and the previous claims, $\max_{q \in Q} d(q, C) \leq 3d^*$. □

Time Complexity:

Only the second algorithm actually needs to be done to find an optimal set of k producers, so per iteration $O(|Q|)$ comparisons are made with each of the $O(k)$ producers in C to determine the farthest $q \in Q'$. $O(|P| - k)$ comparsions are then made between q and the producers not already in C. Thus, the overall running time is $O(k(k|Q| + |P| - k)) = O(k^2|Q| + k|P|)$.

Problem 6: Approximation Algorithm

Claim: The algorithm outputs a sum that is with $(k+3)/(k+1)$ fraction of the optimal output T^*.

Proof. If $n = k$ the partition is optimal, so suppose $n > k$. Let S_1^k and S_2^k be the optimal partition of the k largest elements and let T_k be the sum of the larger subset. Consider the last $c_j, 1 \leq j \leq n$ added to the larger-sum-having subset S' in the partition that minimizes the quantity $\max(\sum_{i \in S_1} c_i, \sum_{i \in S_2} c_i)$. Let T be the sum of S'.

Claim: If the last element added to S', c_j, is one of the k largest elements then $T = T^*$.

Proof. Since c_{k+1} is added to the smaller of S_1^k and S_2^k, S' must be the larger of S_1^k and S_2^k. Therefore, $T = \sum_{i \in S'} c_i = T_k$ does not change for the remainder of the algorithm. Suppose

$T* < T_k$, then the k largest elements would have to be distributed differently between S_1^* and S_2^* to make the sum of the larger subset less than T_k, but this is impossible as it would mean the k partition wasn't optimal. Thus $T* = T_k$ and so $T = T^*$. □

Suppose $j \geq k+1$ and without loss of generality let $S' = S_1$ then $c_j \leq c_k$ and

$$T + \sum_{i \in S_2} c_i \geq 2(T - c_j) \iff T* \geq (\sum_i c_i)/2 \geq T - c_j$$

since

$$\begin{aligned} c_j(k+1)/2 \leq \frac{\sum_{i=1}^{k} c_i + c_j}{2} \leq \frac{\sum_{i=1}^{n} c_i}{2} \leq T^* &\Rightarrow c_j \leq 2T^*/(k+1) \\ &\Rightarrow T - c_j + c_j \leq T^* + c_j \\ &\Rightarrow T \leq \frac{(k+1)T^*}{k+1} + \frac{2T^*}{k+1} \\ &\Rightarrow T \leq \frac{(k+3)T^*}{k+1}. \end{aligned}$$

□

Time Complexity:

There are 2^k possible partitions of the k largest integers into two sets, therefore it takes $O(2^k)$ time to find the optimal partition of these integers. It takes $O(n)$ time to complete the partition by iterating through the remaining $c_i's$, the overall time complexity is $O(2^k + n)$.