Preyasi Gaur Disc 1A

Time: 8:00AM - 9:50 AM

TA: Vincent Li

Computer Science 180

Homework 5

**Question 1**

Algorithm:

* Declaring the variables like such:
  + Let n be the length of s
  + Let x\_cat be the concatenation of x
  + Let y\_cat = x\_cat
* Psuedocode
  + def solution(i, j):   
    if first (i+j) characters of s are an interleaving of i chars of x\_cat and j chars of y\_cat, return true   
    else false
* In this algorithm, our base case is solution (0,0) which will always be True.
* For the sum of indices i+j:
  + Case 1: if the last char s[i+j] is from x\_cat:
    - s[i-1, j] is True
    - s[i+j] is x\_cat[i]
  + Case 2: if the last char s[i+j] is from y\_cat:
    - s[i-1, j] is True
    - s[i+j] is y\_cat[i]
  + If not Case 1 or Case 2, then it just means that the first i+j chars cannot be built from i chars of x\_cat, and j chars of y\_cat 🡪 solution (i, j) == False

Time Complexity Analysis:

* As we need to calculate solution[i, j] for all values of i+j < n, our maximum runtime O(n2)

Proof of correctness:

* We will prove this using induction.
* The base case is (0,0) for which the algorithm returns True.
* Next, as the solution follows a recursive approach, in the sense that the subsequent solutions builds from the previous ones, all the cases will be checked.
* The only way of getting s is through solution [0,n], solution [1, n-1], solution [2, n-2]… solution[n, 0].

**Question 2**

In this question, we can’t straight away apply Djikstra’s (shortest path) algorithm as the edges can also have non positive weights. Thus, we need to come up with a new algorithm.

Algorithm:

* Make a list l1 of size V, with the following intialisations:
  + l1[v] = 0
  + for all other nodes, l1[i] = inf
* We create another list l2 of the same size. We use this l2 to store the shortest paths to the nodes.
  + for all indices, l2[i] = inf
* Now, for V – 1 times:
  + Go through each edge e, and update l1[t] and l2[t] is we are able to find a new shortest path between the nodes.
    - if l1[s] + cost > l2[t]
      * l1[t] = l1[s] + cost
      * l2[t] = l2[s] + e
    - if l1[t] = l1[s] + cost:
      * l2[s] = l2[s]+e

Time Complexity Analysis:

* The for loop runs for V-1 times, over E edges 🡪 O(VE)

Proof of correctness:

* In this case, the shortest path for any two nodes can at max be V-1 length, as anything more will result in a cycle. Here, we know that no negative cycle can exist, and we can remove some edges form the cycle and get a shorter path. This is a contradiction, and no shortest path with a cycle can exist.
* After the first loop across all edges, the shortest path to all vertices that are 1 edge away from s will be found. We will prove this hypothesis by induction. After n iterations, the shortest paths to every node that are n edges away will have been found.
* Base Case: this occurs on the 0 iteration.
* Induction Case: if the shortest distance is calculated for all noes that are n-1 edges away from the source, then in the next step (n), the shortest distance too all the following nodes will have been found.

**Question 3**

Algorithm:

* Define a function solution(p\_idx, num1, A\_count1, A\_count2) having the parameters:
  + p\_idx: Index range for precinct consideration, from 1 up to p\_idx
  + num1: Count of precincts allocated to district 1, implying (p\_idx - num1) are for district 2 and are hence not passed to the function
  + A\_count1: Tally of votes favoring candidate A in district 1
  + A\_count2: Tally of votes favoring candidate A in district 2
* The function solution(p\_idx, num1, A\_count1, A\_count2) has two cases:
  + Appointing the p\_idx precinct to district 1:
    - invoking opt(p\_idx - 1, num1 - 1, A\_count1 - va, A\_count2)`, where va is the vote count for candidate A in precinct p\_idx. The aim is for district 1 to comprise `num1` precincts and to amass `A\_count1` votes for candidate A.
* Assigning the p\_idx precinct to district 2, leading to solution(p\_idx - 1, num1, A\_count1, A\_count2 - va).
* The solution is determined by evaluating solution(n, n/2, A\_count1, A\_count2) for every instance where A\_count1 > mn/4 to guarantee candidate A achieves at least mn/4 votes in each district.
* Initial Conditions:
  + Solution(0,0,0,0) returns True.
  + Any other Solution (x, y, z) configuration yields False.

Time Complexity Analysis:

* Variables under consideration:
  + pidx ranges from 0 to n.
  + num1 falls within 0 to n/2.
  + A\_count1 and A\_count2 each vary from 0 to mn.
* The complexity calculation is as follows:
  + There are O(n4m2) individual subproblems to address.
  + Each subproblem can be concluded in a constant time frame, O(1).

Proof of Correctness:

* We will prove this using induction.
* Base Case: Solution(0,0,0,0) being True, implying that when there are no precincts to assign, the condition is trivially met.
* Inductive Hypothesis: Let's assume that our algorithm works correctly for solution(k, num1, A\_count1, A\_count2). This means that the optimal assignment of precincts up to the k-th one has been found, with num1 precincts in district 1 and A\_count1 and A\_count2 votes for candidate A in district 1 and district 2, respectively. We need to prove that if the algorithm works for k precincts, it will also work for k+1 precincts. The function solution considers two possible scenarios for precinct k+1:
* Assigning precinct k+1 to district 1: This action must preserve the optimality of the vote distribution for candidate A in district 1.
* Assigning precinct k+1 to district 2: Similarly, this action must also maintain the optimality of the distribution for candidate A's votes in district 2.
* For each scenario, the recursive structure of the solution function must ensure that including precinct k+1 does not disrupt the optimal vote allocation achieved thus far. It must be shown that regardless of whether precinct k+1 is added to district 1 or 2, the result leads to the most advantageous outcome under the established voting criteria.

**Question 4**

Algorithm:

* We construct an s-t (source-target) network as described:
* Begin by connecting a source node s to each client node C with an edge capacity of 1, signifying that each client can only be connected to one base station.
* Next, link every client node C to base station nodes B if they are within range. The edge capacities here are set to 1 if in range and are otherwise non-existent. This reflects the possible connection between clients and base stations they can reach.
* Each base station node B is then connected to the target node t, with edge capacities equal to L, representing the maximum number of clients that each base station can serve.
* Implement the Ford-Fulkerson algorithm to determine the maximum flow from s to t.
  + If the absolute value of the maximum flow `|f|` is less than `n`, it indicates that not all clients can be connected to exactly one base station since there's insufficient capacity to accommodate all connections.

Time Complexity Analysis:

* Constructing the network takes O(nk) time, as each client C can potentially connect to k base stations.
* The time complexity for the Ford-Fulkerson algorithm is O(|f|e), where |f| is the flow value and e is the number of edges.
* For e, we calculate the following:
  + There are n source-client edges.
  + Client-station edges number up to O(nk) because each client can connect to up to k stations.
  + Station-target edges are exactly k.
* We know that the maximum flow |f| is at most Lk because that's the total outgoing capacity from all stations to the target node t.
* Hence, the time complexity for running Ford-Fulkerson is O(nLk2).

Proof of correctness:

* Load: Each station can only handle up to L clients, ensured by the capacity of the edges from stations to the target node t.
* Range: Each client can only connect to base stations within range, which is guaranteed by the structure of our network and the capacities of the edges from clients to stations.
* Exclusive Connection: Each client must be connected to exactly one base station, which is confirmed by two sub-conditions:
  + Clients cannot be connected to more than one base station due to the incoming flow to clients being capped at one. This is enforced by the edge capacity from s to each client and the flow conservation principle within the network.
  + Each client must be connected to at least one base station, which is determined during the flow calculation. If a client is not connected, it will be reflected in the final flow value being less than the total number of clients n.

By ensuring all these conditions, the solution design effectively translates the problem constraints into a network flow model, where the Ford-Fulkerson algorithm can find an optimal assignment of clients to base stations.

**Question 5**

Algorithm:

* We can solve this problem using the max-flow min-cut theorem by creating a flow network where each client and base station is represented as a node.
* Create a source node s and a sink node t.
* Connect s to each client node C\_i with an edge of capacity 1. This ensures that each client can only be connected to one base station.
* Connect each client node C\_i to base station node B\_j if the distance between client C\_i and base station B\_j is less than or equal to r. The capacity of each of these edges is 1.
* Connect each base station node B\_j to t with an edge of capacity L. This ensures that no more than L clients can be connected to a single base station.
* Run the Ford-Fulkerson algorithm to find the maximum flow in this network.
* If the maximum flow is equal to the number of clients n, then each client can be connected to a base station while satisfying the range and load parameters.

Proof of Correctness:

* Base Case: The network starts with no flows, which is trivially correct as no clients are connected, and no base station is overloaded.
* Inductive Step: Each flow added represents a connection from a client to a base station. Because of the construction of the network and the capacities of the edges, no client can be connected to more than one base station, and no base station can have more than L clients.
* When Ford-Fulkerson runs, it iteratively finds augmenting paths from s to t. Because all capacities are integers (1 or L), the algorithm will incrementally increase the flow until no more augmenting paths can be found, ensuring that each client is connected to a base station if such a configuration is possible.

Time Complexity Analysis:

* Building the Network: O(nk) because we must check for a possible edge between each of the n clients and k base stations.
* Ford-Fulkerson Algorithm: The time complexity is O(Ef) where E is the number of edges in the network and f is the maximum flow. Here, the maximum flow f is bounded by n (since each client can contribute at most 1 to the flow), and the number of edges E is at most n + nk + k (n edges from source to clients, at most nk edges from clients to base stations, and k edges from base stations to the sink). Hence, the time complexity is O(nk \* n) in the worst case.
* Overall, the time complexity of the algorithm is dominated by the Ford-Fulkerson algorithm, making it O(n2k). This is a polynomial-time algorithm, given that n and k are the inputs to the problem.

**Question 6**

Algorithm:

* Initialize two arrays, high[i] and low[i], of the same length as the input sequence where i ranges from 0 to n-1. Here, n is the length of the input sequence. high[i] will store the length of the longest alternating subsequence ending at index i where the last element is higher than its predecessor. Similarly, low[i] will store the length of the subsequence where the last element is lower.
* Set high[0] and low[0] to 1 because the first element by itself is a subsequence of length 1.
* Iterate through the sequence from the second element to the end:
  + For each element at index i:
    - Set low[i] equal to high[i-1] + 1 if the current element is less than the previous element; otherwise, carry forward the maximum low value found so far (low[i] = low[i-1]).
    - Set high[i] equal to low[i-1] + 1 if the current element is greater than the previous element; otherwise, carry forward the maximum high value found so far (high[i] = high[i-1]).
* The length of the longest alternating subsequence is the maximum value in either `high` or `low` arrays after the final iteration.

Time Complexity Analysis:

* The time complexity of the algorithm is O(n), where `n` is the number of elements in the input sequence.
* This is because we traverse the input sequence only once, and at each step, we perform a constant number of operations.

Proof of Correctness:

We will prove this using induction.

* Base Case: For a sequence of one element, the longest alternating subsequence is the element itself, which makes the base case correct.
* Inductive Step: Assume that high[i] and low[i] correctly compute the longest subsequences for all elements up to index i. When we move to the next element at index i+1, we decide based on the comparison with the element at index i. If the current element is lower, it can extend a subsequence that last went higher, and vice versa. Thus, the length of the longest alternating subsequence up to index i+1 is correctly computed based on the subsequence lengths up to index i.
* Since the base case is correct, and the inductive step proves that if the subsequence lengths are correct up to index `i`, they will be correct up to index `i+1`, by induction, the algorithm correctly computes the longest alternating subsequence for the entire sequence.