E0-225: Design & Analysis of Algorithms

Manish Kumar (21044)
M.Tech., Quantum Tech.

Assignment-I

August 29, 2023

August 29, 2023

Problem 1

Describe and analyze an efficient algorithm to determine whether a given set of male and female preferences admits a unique stable matching.

Solution.

As discussed in class, the Gale-Shapley algorithm (GSA) guarantees optimal matching using the 'deferred acceptance strategy'. But there could be several such optimal matching. GSA guarantees to output one of them.
As pointed out in the hint-
Case(I): proposer→ male and proposee→ female $\implies$ males get the best possible match while females get the worst.
Case(II): proposer→ female and proposee→ male $\implies$ females get the best possible match while males get the worst.
Now, we can deduce that if the best possible match and the worst possible match for males (or females) coincide, then there is only one matching pair possible. Hence, there exists unique stable matching.

Idea: We apply GSA algorithms twice. In the first case, males are proposers. Then, in the second case, females are proposers. If, in both cases, the output is the same, then declare unique stable matching. Otherwise, output that unique matching doesn't exist.

- Input= preference list of males and females
- Output= {YES, NO}

Psuedo code:

$result\_1 \leftarrow GSA(male\_as\_proposer)$
$result\_2 \leftarrow GSA(female\_as\_proposer)$
if $result\_1 == result\_2$ **then**
    $output \leftarrow YES$
else
    $output \leftarrow NO$

Runtime analysis: As discussed in class, running GSA once has the time complexity of $\mathcal{O}(n^2)$. A male has to make a maximum 'n' proposal to the females. This iterates over all 'n' males. Hence, the worst-case run time (asymptotically) scales quadratic in 'n'. In the above algorithm, GSA runs twice. Hence, it still has a time complexity of $\mathcal{O}(n^2)$.

Problem 2

Describe and analyze an efficient algorithm to compute the smallest interval cover of X. Also provide proof of correctness.

Solution. The interval scheduling problem (ISP) discussed in class gives some ideas to tackle this problem. In ISP, the idea is to maximize the number of elements in the solution set, yet none clashes.
In some sense, this problem has its requirement complementary to ISP. The elements of the solution set should overlap while keeping the cardinality of the set minimal.
Algorithm:

- Step-1: Sort the intervals as per the 'start first' criteria.
- Step-2: Record the minimum of array $L[1,..,n]$ (say, L_{min}) and maximum of array $R[1,...,n]$ (say, R_{max}). This is to get an idea of the maximum time spread of the complete set.
- Step-3: Begin with the interval having its leftmost (or stating value) $= L_{min}$. (If there is more than one such interval, pick the one that ends at the latest.) Add it to the solution set. Level it as temporary variable 'temp'.
- Step-4: Among all the intervals that clash with 'temp', include that interval in the solution set that finishes lastest. Now update 'temp'.
- Iterate the above procedure until the end point of 'temp' is R_{max}.

Note: It is assumed that 'X' do admit an interval cover. If there is no interval cover at all, then the above algorithm might not work as intended.

Runtime analysis: Time complexity of sorting goes as $\mathcal{O}(n \cdot log(n))$. A single loop iterating over all the elements is another requirement. Hence, Asymptotic runtime is $\mathcal{O}(n \cdot log(n))$.
Proof of correctness: We can see it employs greedy strategy. The argument for correctness follow similar to interval scheduling problem. Suppose there exist some other algorithm X that produces a solution set having lesser cardinality than the above algorithm (Say, A). It possible if the algorithm X has taken included an interval that was not locally optimal. Or, there was a longer interval included by algorithm X but not by the algorithm A. This conflicts with the assumption that algorithm A was taking locally optimal decision. It implies the existence of algorithm X is not correct.

Problem 3

Scheduling unit-time tasks with deadlines. To show the suggested algorithm always outputs the optimal answer.

Solution. Algorithm suggested:

- Step-1: Initialize an empty array of time slots of length 'n'. The ith element of the array is a unit-length slot of time that finishes at time 'i'.
- Step-2: Short the task in order of monotonically decreasing penalty.
- Step-3: For scheduling the task 'j', check if there exists any empty slot at or before 'j"s deadline d_j that is still empty. Then, assign 'j' to the latest among these slots.
- Step-4: List all the intervals that clash with 'temp'. Now include that interval in the solution set, which started first, and its starting point is less than the starting point of 'temp'. Now update 'temp'.
- Iterate the above procedure until the starting point of 'temp' is L_{min}.

Proof of optimality: This is akin to greedy strategy. We can use a similar notion like 'exchange argument' to show its optimality.

- Assume the minimal penalty incurred by some other algorithm (say, X) is P' while for the above algorithm (say, G) yields P penalty (where $P > P'$).
- Suppose the scheduling sequence as per algorithm X is $(x_1, x_2, ...)$ while algorithm G yields $(g_1, g_2, ...)$.
- The key thing to note is that the penalty cost of G is higher than X if and only if G missed placing (at least) one of the jobs with a higher penalty on or before a deadline. Rather, G prioritized some other job with a lesser penalty. But we know G always prioritize maximum penalty task over lesser one. (The prioritization argument comes from the description of G mentioned in the problem statement. It is like greedy.)
- This implies inconsistency with the very own working of algorithm G. Hence, Our assumption of the existence of algorithm X must be wrong. Thus, the algorithm G is as good as other optimal algorithms to solve this problem.

Problem 4

Describe and analyse an efficient algorithm to compute the minimum-weight feedback edge set of a given edge-weighted edge graph.

Solution. By the definition of feedback edge set, we can sense that it is the edge complement of the spanning tree of a graph. Because if we add any extra edge to the spanning tree, it is destined to have a cycle. Extra edge refers to those edges of a graph that is not part of spanning tree.

In the case of a minimum spanning tree, the remaining edges will form a maximum weight feedback edge set. This is because a minimum spanning tree avoids picking the maximum weight edge among any possible cycle in a graph induced by adding an extra edge. (If it does so, we can always come up with another spanning tree that weighs less. This is just by removing the maximum weight edge and adding the second maximum edge to the spanning tree.)

Hence, from the above argument, we can infer that the minimum weight feedback edge set is the edge complement of the maximum spanning tree.

Algorithm: To find maximum spinning tree, we use the variant of Borůvka. We add all the safe edges and recurse. But now, safe edge is redefined as maximum-weight edge instead of minimum-weight edge. Once we have constructed the maximum spanning tree, we take its edge complement to get the final answer.

- Start with the trivial forest $F = (V, \Phi)$. It include all the vertex but no edges.
- The algorith requires construction of an array $safe[1..V]$. We do it by taking $safe[i]$ as the maximum-weight edge with one endpoint in the ith component of F, by an exhaustive search over each edge in graph G.
- The definition of *useless* edge remain the same. It is not an edge of F, but both of its endpoints lie in the same component of F.
- Then we keep adding safe edges similar to original Borůvka.

Runtime analysis: The runtime is similar to Borůvka for MST. It iterates over all the edges while keep adding safe edges to the initialized list. The iteration over all the edges requires $\mathcal{O}(E)$ time. While the check for safe edge within each iteration requires $\mathcal{O}(log(V))$. The reason behind logarithm is the same. The decrease in the components of F is at least half as iteration progresses. Hence, over all complexity is $\mathcal{O}(E) \cdot log(V)$.

Problem 5

Second smallest spanning tree.

- Part-1: The second smallest spanning tree (second-MST) is unique if all weights are unique.
- Part-2: If T is the MST of G. Then Prove that G contains edge $e \in T$ and $e' \notin T$ such that $T - e - e'$ is the second smallest spanning tree.
- Part-3: Describe and analyse an efficient algorithm to find the second smallest spanning tree of G.

Solution.

- Part-1: The statement is False. Second-MST is not neccesarly be unique. We can give a counterexample to support the claim. [See the picture at the last page]
- Part-2: Basically we need to show that second-MST exactly have one edge that is not the part of MST. Rest edges are same as of MST. We use the fact that all spanning tree must have same number of edges, and MST is always unique in the given context. Now we show it forces a second-MST to choose exactly one edge not in MST but the increase in the overall weight should be as minimum as possible.
  Now we need to rule out the case of removing two edge from MST and choose two other extra edge to form second-MST. This is done by contradiction by assuming such strategy work and then showing that it leads to conclusion that there can exist more than one MST.
- Part-3: One algorithm to generate such graph would be to choose any edge of the graph that is not part of the tree. Add it to the tree. This will make a cycle in the graph. Now inspect the cycle and find the edge that has second highest weight. Now remove that edge. We get a tree again but it is not the MST anymore. It is second-MST instead.