CSCI 570 - Summer 2021 - Homework 2 Solutions

III. GRADED PROBLEMS

1) Sort A and B in the same order (either both ascending or both descending). This takes $\mathcal{O}(n\log(n))$. What's important is that the largest element of A is matched with the largest of B, the second-largest of each are matched, and so on (greedy solution). We need to show that this solution is optimal.

   For the purpose of proof, fix the order of A in an ascending order, i.e., $a_1 \leq a_2 \leq ... \leq a_n$; we will consider all solutions in terms of how their B arranged relative to this A. For any arbitrary solution (could be an optimal solution): $\{b_i\}$, where b_i is the element of B matched with a_i in this solution. Suppose there exists $b_i < b_j$ such that $i > j$. We show that by swapping these two elements the resulting solution is greater than the previous one which is a contradition.

$$\frac{a_i^{b_j} \times a_j^{b_i}}{a_i^{b_i} \times a_j^{b_j}} = \frac{a_i^{b_j - b_i}}{a_j^{b_j - b_i}} > 1 \tag{1}$$

2) Construct a min-heap of the k sublists, that is each sublist is a node in the constructed min-heap. When comparing two sublists, compare their first elements (that is their minimum elements). The creation of this min-heap will cost $\mathcal{O}(k)$ time.

   Then we run the extract min algorithm and extract the minimum element from the root list. Then update the root sublist and heapify the min-heap according to the new minimum element in the root sublist. (If the root sublist becomes empty during this step, take any leaf sublist and make it the root and then heapify). Each extraction takes $\mathcal{O}(\log k)$ time. Since we extract n elements in total, the running time is $\mathcal{O}(n\log k + k) = \mathcal{O}(n\log k + k)$ (since $k < n$).

3) Suppose that the minimum weighted edge is not on the MST. We add this edge to the MST, which results in a cycle. Removing an edge other than the minimum edge on the cycle leads to a tree with a smaller total weight which is a contradiction.

4) Assume one needs two replacements. If one of the replacements decreases the weight then doing this replacement first would produce a tree lighter than the minimal spanning tree. If both replacements increase weight, then there would be a tree with weight strictly between the best and the second best tree, by using one replacement only.

5) Sort cows in ascending order of $S_i + W_i$ from top to bottom

The proof is similar to the proof we did for the scheduling problem to minimize maximum lateness. We first define an inversion as a cow i with higher $(W_i + S_i)$ being higher in the tower compared to cow j with lower $(W_j + S_j)$. We can then show that inversions can be removed without increasing the maximum risk value time. We then show that given an optimal solution with inversions, we can remove inversions one by one without affecting the optimality of the solution until the solution turns into our solution.

a) Inversions can be removed without increasing the risk value. Remember that if there is an inversion between two items a and b, we can always find two adjacent items somewhere between a and b so that they have an inversion between them. Now we focus on two adjacent cows (one standing on top of the other) who have an inversion between them, e.g. cow i with higher $(W_i + S_i)$ is on top of cow j with lower $(W_j + S_j)$. Now we show that scheduling cow i before cow j is not going to increase the maximum risk value of the two cows i and j. We do this one cow at a time:

- By moving cow j higher we cannot increase the risk value of cow j
- cow j but since cow i has a higher total weight and strength than cow j, the risk value of cow i (after removing the inversion) will not be worse than the risk value for cow j prior to removing the inversion:
- Risk value of cow j before removing the inversion: W_i+ (weight of remaining cows above cows i and j) $-S_j$
- Risk value of cow i after removing the inversion: W_j+ (weight of remaining cows above cows i and j) $-S_i$
- Since $W_i + S_i >= W_j + S_j$, if we move S_i and S_j to the opposite sides we get: $W_i - S_j >= W_j - S_i$. (weight of the remaining cows above the two cows i and j does not change). In other words, the risk value of cow j before removing the inversion is higher than that of cow i after removing the inversion

b) Since we know that removing inversions will not affect the maximum risk value negatively, if we are given an optimal solution that has any inversions in it, we can remove these inversions one by one without affecting the optimality of the solution. When there are no more inversions, this solution will be the same as ours, i.e. cows sorted from top to bottom in ascending order of weight+strength. So our solution is also optimal.

IV. Practice Problems

1) Solve Kleinberg and Tardos, Chapter 3, Exercise 3.

   We run the algorithm described in Section 3.6 (bottom of page 106) with the following modification. If the input G is a DAG, then at each iteration of the algorithm, there exists a vertex with no incoming edges and the algorithm finds a topological ordering thereby establishing that the input G is indeed a DAG. Hence the only way for the algorithm to be stuck is if at a certain iteration all the vertices have at least one incoming edge. In this case, pick an arbitrary vertex v_1 and follow one of its incoming edges to a predecessor vertex (say v_2). Now follow one of v_2's incoming edges to go to one of its predecessors (say v_3) and so on. We can repeat this step as often as we want since every vertex has an incoming edge. Since the graph has at most n vertices, after at most n steps we have to revisit a vertex (say $v_k = v_i$ for some $k > i$). Clearly $v_i, v_i + 1, ..., v_k$ forms a cycle.

2) Solve Kleinberg and Tardos, Chapter 3, Exercise 6.

   Assume that G contains an edge $e = (x, y)$ that does not belong to T. Since T is a DFS tree and (x, y) is an edge of G that is not an edge of T, one of x or y is ancestor of the other. On the other hand, since T is a BFS tree if x and y belong to layer L_i and L_j respectively, then i and j differ by at most 1. Notice that since one of x or y is an ancestor of the other, we have that $i \neq j$ and hence i and j differ by exactly 1. However, combining that one of x or y is ancestor of the other and that i and j differ by 1 implies that the edge (x, y) is in the tree T. It contradicts the assumption that $e = (x, y)$ that does not belong to T. Thus G cannot contain any edges that do not belong to T.

3) Solve Kleinberg and Tardos, Chapter 4, Exercise 21.

   To do this, we apply the cycle property nine times, i.e. we perform BFS until we find a cycle in the graph G and then we delete the heaviest edge on this cycle. We have now reduced the number of edges in G by one, while keeping G connected and not changing the identity of the minimum spanning tree (again by the cycle property). If we do this nine times, we will have a connected graph H with $n - 1$ edges and the same minimum spanning tree as G. But H is a tree and thus it has to be the minimum spanning tree of G. The running time of each iteration is $O(m + n)$ for the BFS and

subsequent check of the cycle to find the heaviest edge. Since $m < n + 8$ and there are a total of nine iterations, total running time is $O(n)$.

4) Solve Kleinberg and Tardos, Chapter 4, Exercise 22

Consider the following counterexample. Let $G = (V; E)$ be a weighted graph with vertex set $V = \{v_1, v_2, v_3, v_4\}$ and edges (v_1, v_2) (v_2, v_3) (v_3, v_4) (v_4, v_1) of cost 2 each and an edge (v_1, v_3) of cost 1. Then every edge belongs to some minimum spanning tree, but a spanning tree consisting of three edges of cost 2 would not be minimum.

5) (a) Algorithm: Denote the coins values as $c1 = 1,\ c2 = 5,\ c3 = 10,\ c4 = 25$.

   a) if $n = 0$, do nothing but return.
   b) Otherwise, find the largest coin c_k, $1 \leq k \leq 4$, such that $c_k \leq n$. Add the coin into the solution-coin set S.
   c) Substract c_k from n, and repeat the steps 1) and 2) for $n - c_k$

   For the proof of optimality, we first prove the following claim: Any optimal solution must take the largest c_k, such that $c_k \leq n$. Here we have the following observations for an optimal solution:

   a) Must have at most 2 dimes; otherwise we can replace 3 dimes with quarter and nickel.
   b) If 2 dimes, no nickels; otherwise we can replace 2 dimes and 1 nickel with a quarter.
   c) At most 1 nickel; otherwise we can replace 2 nickels with a dime.
   d) At most 4 pennies; otherwise can replace 5 pennies with a nickel.

   Correspondingly, an optimal solution must have

   a) Total value of pennies: ≤ 4 cents.
   b) Total value of pennies and nickels: $\leq 4 + 5 = 9$ cents.
   c) Total value of pennies, nickels and dimes: $\leq 2 \times 10 + 4 = 24$ cents.

   Therefore,

   a) If $1 \leq n < 5$, the optimal solution must take a penny.
   b) If $5 \leq n < 10$, the optimal solution must take a nickel; otherwise, the total value of pennies exceeds 4 cents.
   c) If $10 \leq n < 25$, the optimal solution must take a dime; otherwise, the total value of pennies and nickels exceeds 9 cents.
   d) If $n \geq 25$, the optimal solution must take a quarter; otherwise, the total value of pennies, nickels and dimes exceeds 24 cents.

Comparing with the greedy algorithm and the optimal algorithm, since both algorithms take the largest value coin c_k from n cents, then the problem reduces to the coin-changing of $n - c_k$ cents, which, by induction, is optimally solved by greedy algorithm.

(b) Coin combinations $= \{1,\ 15,\ 20\}$ cents coins. Consider this example $n = 30$ cents. According to the greedy algorithm, we need 11 coins: $30 = 1 \times 20 + 10 \times 1$; but the optimal solution is 2 coins $30 = 2 \times 15$.