Algorithm Design and Analysis (Fall 2021)
Final Exam Solution

1. (25 points) Consider the following variant of Kruskal Algorithm. Does it correctly find a maximum spanning tree on an undirected weighted graph? If so, prove its correctness. If not, give a counterexample ($G = (V, E), w : E \to \mathbb{R}^+$) showing that it fails to find a maximum spanning tree.

   - Sort the edges by the weight *descending* order, and initialize $S \leftarrow \emptyset$.
   - For each edge e in the order, if adding e to S does not creat a cycle, then add it to S.
   - Terminate when S forms a spanning tree.

   The algorithm correctly finds a maximum spanning tree.

   Proof. Let S_i be the tree at the i-th iteration. We will prove by induction that S_i is a part of an optimal solution (a maximum spanning tree) for all i.

   Base Step: $S_0 = \emptyset$ is clearly a part of an optimal solution.

   Inductive Step: Suppose $S_i \subseteq S^*$ for some maximum spanning tree S^*. Let $S_{i+1} = S_i \cup \{e\}$. If $S_{i+1} \subseteq S^*$, we are done. Suppose $S_{i+1} \not\subseteq S^*$. Then $S^* \cup \{e\}$ contains a cycle C. We discuss two cases.

   Case 1: there exists $e' \in C \setminus S_{i+1}$ with $w(e') \leq w(e)$. Then, $S^{**} = S^* \cup \{e\} \setminus \{e'\}$ is a spanning tree with $w(S^{**}) \geq w(S^*)$. S^{**} is also optimal, and $S_{i+1} = S_i \cup \{e\} \subseteq S^{**}$.

   Case 2: $w(e') > w(e)$ for all $e' \in C \setminus S_{i+1}$. For any $e' \in C \setminus S_{i+1}$, we have $e' \in C \subseteq S^*$ and $S_i \subseteq S^*$ (induction hypothesis), so $S_i \cup \{e'\} \subseteq S^*$. This implies $S_i \cup \{e'\}$ contains no cycle. Then the algorithm should not choose e at the $(i+1)$-th iteration, as choosing e' with larger weight does not create a cycle. We have a contradiction. $\square$

   Grading Rubrics:

   - (-25 points) The answer is wrong (claiming that the algorithm fails to find a maximum spanning tree) without a justification (or with a justification that does not make sense).
   - (-20 points) The answer is wrong (claiming that the algorithm fails to find a maximum spanning tree), but some justification is given.
   - Some points are taken off accordingly for a right answer with a proof that is not rigorous, depending on the quality of the proof.

2. (25 points) Suppose we want to allocate m different items to n agents. Let M be the set of items and $N = \{1, \ldots, n\}$ be the set of agents. Each agent i has a *demand set* $N(i) \subseteq M$ which describes the subset of items wanted by agent i. In addition, each agent i can be allocated at most $c(i) \in \mathbb{Z}^+$ items. Design a polynomial time algorithm that allocates the maximum number of demanded items. That is, your algorithm takes $(\{N(i), c(i)\}_{i=1,\ldots,n})$ as the input and outputs an allocation $(A_1, \ldots, A_n)$ maximizing $\sum_{i=1}^n |A_i|$ subject to that

   - $A_i \subseteq N(i) \subseteq M$ and $A_i \cup A_j = \emptyset$ for any $i \neq j$, and
   - $|A_i| \leq c(i)$.

   Prove the correctness of your algorithm and analyze its running time.

   The problem can be converted to a max-flow problem as follows. Create a source s and a sink t. For each agent i, create a vertex a_i and an edge (s, a_i) with capacity $c(i)$. For each item j, create a vertex v_j and an edge (v_j, t) with capacity 1. For each pair (i, j) with $j \in N(i)$, create an edge (a_i, v_j) with capacity 1. This completes the description of the conversion.

   To find an optimal allocation, we first find a maximum flow on the graph constructed above. We use Dinic's algorithm (or Edmond-Karp algorithm) to find the maximum flow to make sure the flow found is integral. Finally, output the allocation $(A_1, \ldots, A_n)$ such that $j \in A_i$ if the flow on the edge (a_i, v_j) has value 1.

   To show the correctness of the algorithm, first of all, notice that the flow output by the algorithm is *integral*, as all capacities are integers. The remaining part of the correctness proof is straightforward. The flow conservation constraint ensures that each item is allocated to at most one agent, and the number of items received by agent i does not exceed $c(i)$. In addition, A_i never contain an item j that is not in $N(i)$, as (a_i, v_j) is not an edge in our graph.

   The time complexity for Dinic's algorithm is $O(|V|^2|E|)$ for a graph $G = (V, E)$. Here, we have $m + n + 2 = O(m + n)$ vertices and at most $mn + m + n = O(mn)$ edges. The time complexity for our algorithm is therefore $O(mn(m + n)^2)$. Better analysis of the time complexity is possible. However, students get the full credit if the time complexity is polynomial and the analysis of the time complexity is correct.

   Grading Rubrics:

   - 10 to 15 points are taken off if an incorrect algorithm (e.g., based on greedy or dynamic programming) is presented. The number of the points being taken off depends on the quality of the solution.
   - (-5 points) No correctness proof.

- (-3 points) In the proof of the correctness, the important point about the flow integrality is missing.
- (-5 points) No time complexity analysis
- (-2 points) The time complexity is in terms of $|V|$ and $|E|$ instead of m and n.

3. (25 points) Given a $m \times n$ non-negative integer matrix $A \in \mathbb{Z}_{\geq 0}^{m\times n}$, suppose we want to walk from the entry $(1,1)$ to the entry (m,n). In each step, we can walk from (i,j) to either $(i+1,j)$ or $(i,j+1)$. Design a polynomial time algorithm that finds a valid walk which maximizes the sum of the visited entries. Prove the correctness of your algorithm and analyze its time complexity.

   Let $S(i,j)$ be the maximum possible sum of the visited entries in the walk from $(1,1)$ to (i,j). We have the following recurrence relation.

$$S(i,j) = \begin{cases} \sum_{j'=1}^{j} A_{ij'} & \text{if } i = 1 \\ \sum_{i'=1}^{i} A_{i'j} & \text{if } j = 1 \\ \max\{S(i-1,j), S(i,j-1)\} + A_{ij} & \text{otherwise} \end{cases}.$$

   A dynamic programming based algorithm can be easily designed based on this. The correctness is straightforward by induction, and the time complexity is $O(mn)$.

   Notice that Dijkstra's algorithm can also be adapted to this question, but students need to justify its correctness.

   Grading Rubrics:

   - (-15 points) Incorrect algorithm.
   - (-5 points) Invalid analysis of correctness.
   - (-5 points) Incorrect analysis of time complexity.

4. (25 points) Consider the *Knapsack* problem. You have a set of items $N = \{1, \ldots, n\}$ and a capacity constraint $K \in \mathbb{Z}^+$. Each item i has a *weight* $w_i \in \mathbb{Z}^+$ and a *value* $v_i \in \mathbb{Z}^+$. The objective is to find a subset of items $S \subseteq N$ with maximum total value $\sum_{i \in S} v_i$ subject to the capacity constraint $\sum_{i \in S} w_i \leq K$. You can assume $w_i \leq K$ for each item i.

   (a) (10 points) Prove that Knapsack is NP-hard.

   (b) (10 points) Consider a special case of this problem where we have $w_i = v_i$ for each item i. Prove that the floowing greedy algorithm gives an 0.5-approximation: iteratively pick an item with maximum $w_i = v_i$ until no more item can be added to S.

   (c) (5 points) Provide a tight example showing that the algorithm in (b) cannot do better than 0.5-approximation.

(a) We reduce the problem from SubsetSum+. Given a SubsetSum+ instance $(\{a_1, \ldots, a_n\}, k)$, we construct a Knapsack instance as follows. For each a_i, construct an item i with weight and value being $w_i = v_i = a_i$. The capacity constraint is set to $K = k$.

If the SubsetSum+ instance is a yes instance, we have a collection S of integers with sum k. In the Knapsack instance, the items corresponding to S satisfies the capacity constraint, and the total value is k.

If the SubsetSum+ instance is a no instance, we will show that any collection of items S satisfying the capacity constraint has total value *strictly less than* k. Suppose this is not the case. First, notice that the total value can at most be k, since each item's value equals to its weight. Thus, having a total value *not* strictly less than k implies the total value is *exactly* k. This implies, in the SubsetSum+ instance, there is a collection of integers with sum exactly k. The SubsetSum+ instance is therefore a yes instance, which is a contradiction.

(b) If the algorithm terminates by selecting all the n items, the solution output is clearly optimal. From now on, we consider the case where there are items that are not selected. Suppose the algorithm outputs S and the last item selected by the algorithm is t. We have $v_j = w_j < v_t = w_t$ for any $j \notin S$ by the nature of the greedy algorithm. Moreover, we must have $k - \sum_{i \in S} w_i = k - \sum_{i \in S} v_i < v_t = w_t$, for otherwise more items should be added to S and the algorithm should not have terminated. Therefore, $v_t + \sum_{i \in S} v_i > k$. Since the sum contains the term v_t, we have $\sum_{i \in S} v_i \geq v_t$, so $\sum_{i \in S} v_i > 0.5k$. Since k is clearly an upper bound to the optimal value, this is a 0.5-approximation algorithm.

(c) Consider the following Knapsack instance with three items such that $v_1 = w_1 = T$, $v_2 = w_2 = T$ and $v_3 = w_3 = T + 1$, and $K = 2T$, where T is a large integer. The greedy algorithm will select item 3 and terminate, and the total value is $T + 1$. The optimal solution is to select items 1 and 2, and the total value is 2. The greedy algorithm in this case provides only a $\frac{T+1}{2T}$-approximation. This can be arbitrarily close to 0.5 by having $T \to \infty$. Thus, this example shows that the greedy algorithm can do no better than a 0.5-approximation.

Grading Rubrics:

(a) Some points are taken off if the reduction is incorrect or the proof is incomplete. The number of points being taken off depends on how much sense the solution make, and the quality of the proof.

(b) Some points are taken off if the proof is not rigorous. The number of points being taken off depends on the quality of the solution.

(c) 3 points are taken off for an incorrect example with a mostly correct idea. 5 points are taken off if the example given is completely incorrect.