Solutions to Homework 9

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't do last week's homework, choose a problem from it that looks challenging to you, and in a few sentences, explain the key ideas behind its solution in your own words.

Solution: My solution to 6.e. on the previous homework was as follows:

> **Solution:** If $\phi \in$ 3SAT, then there exists a submultiset of A whose sum of decimal digits matches s. Thus, $(A, s) \in$ SUBSET-SUM.

This solution is fundamentally deficient, stemming from my approach of the problem. My solution for the other direction of the correctness condition was similarly simple, but was partially correct. I should have noted that the construction of t_i and f_i implies that each clause has at least one true literal, and thus is satisfied, and then constructed a boolean value to correspond to whether or not t_i or f_i is in I. This would lead to a conclusion that the first m significant digits in I sum to 1, 2, or 3 with just t_i and f_i numbers.

2. **Approximation warm-up.**

(2 pts) (a) Suppose you have an instance of a minimization problem whose optimal value is 6. Specify the range of values that a 3/2-approximation algorithm could output when run on this instance.

Solution: For a minimization problem with an optimal value of 6, a $\frac{3}{2}$-approximation algorithm outputs a value at most $\frac{3}{2} \cdot 6 = 9$. The output is in the range $[6, 9]$.

(2 pts) (b) Suppose you have an instance of a maximization problem whose optimal value is 25. Specify the range of values that a 2/5-approximation algorithm could output when run on this instance.

Solution: For a maximization problem with an optimal value of 25, a $\frac{2}{5}$-approximation algorithm outputs a value at least $\frac{2}{5} \cdot 25 = 10$. The output is in the range $[10, 25]$.

3. **Revisiting Times Square.**

Kate is driving her limousine through Manhattan on New Year's Eve, heading toward Times Square. Unlike her competitor Vikram (from Homework 4), Kate doesn't care about profits—she just wants to enjoy the journey and pick up as many tour groups as possible on her way to Times Square, stopping at each intersection to wave hello.

However, Kate refuses to backtrack or repeat any intersections–she insists on a simple path from her starting location to Times Square.

Formally, we define the following language:

$$\text{K-PATH} = \left\{ (G = (V, E), s, t, k) : \begin{array}{r} G \text{ has a simple path from } s \text{ to } t \text{ that} \\ \text{visits at least } k \text{ vertices in } V \end{array} \right\}.$$

(8 pts) (a) Show that K-PATH is NP-hard.

Solution: Reduce from Hamiltonian-Path: Given an instance (G', s', t') asking whether there is a Hamiltonian path from s' to t' in G', construct $\big(G',\ s',\ t',\ |V'|\big)$. The mapping is immediate by setting $k = |V'|$; so it clearly runs in polynomial time.

- ($\Rightarrow$) If G' has a Hamiltonian path from s' to t', then G' has a simple path of length $|V'|$, thus $\big(G', s', t', |V'|\big) \in$ K-PATH.
- ($\Leftarrow$) If $\big(G', s', t', |V'|\big) \in$ K-PATH, then there is a simple path of length $|V'|$ from s' to t', which is a Hamiltonian path in G'.

Hence Hamiltonian-Path $\leq_p$ K-PATH, proving K-PATH is NP-hard.

(8 pts) (b) Given an efficient decider D_{KP} for K-PATH, design and analyze (both correctness and runtime) an efficient algorithm that given $(G = (V, E), s, t)$, and k^*, the length of the longest path from s to t, finds an actual simple path from s to t that goes through k^* vertices. The path should be returned as a set of edges (the edges do **not** have to be ordered). You may assume that $\{s, t\} \subseteq V$. If no such path exists, output $\emptyset$.

Solution: Given a polynomial-time decider D_{KP} for K-PATH and an input $\big(G = (V, E)$, s, t, $k^*\big)$:

1. Initialize a set of edges $H = \emptyset$. Let $u = s$ and let $\ell = k^*$.
2. While $\ell > 1$ and $u \neq t$:
   (a) For each neighbor v of u in G, temporarily remove (u, v) from G. Query D_{KP} on $\big(G, s, t, \ell\big)$.
   (b) If $\big(G, s, t, \ell\big)$ remains a "yes" instance, keep (u, v) removed. Otherwise, restore (u, v) to G, add (u, v) to H, set $u \leftarrow v$, decrease $\ell \leftarrow \ell - 1$, and break from the neighbor loop.
3. If $u = t$, output H. Otherwise, output $\emptyset$.

Each time removing an edge preserves a "yes" answer, it is not on every valid path of length ℓ; each time an edge must be restored, it is part of all such paths and is added to H. Continuing until either reaching t or exhausting the budget ℓ yields the largest simple path through exactly ℓ vertices. If no path exists, $\emptyset$ is returned, and thus the algorithm is correct. Each edge removal or restoration is checked by one call to D_{KP}. Since each vertex is moved to at most once along the constructed path, the total number of queries is polynomial in $|E|$. Thus, the overall procedure is efficient.

(12 pts) 4. **Finding the colors through SAT.**

Recall the languages:

$$\text{3-COLORING} = \{G : G \text{ is a graph and is 3-colorable}\}$$
$$\text{SAT} = \{\phi : \phi \text{ is a satisfiable Boolean formula}\}$$

In this problem, you will show that if we have an efficient decider for SAT, then we can efficiently solve the 3-COLORING search problem.

First, consider the following PTMR from 3-COLORING to SAT without using the Cook-Levin Theorem.

Input: A graph G
Output: A Boolean formula ϕ

function $f(G = (V, E))$
    Initialize an empty boolean formula ϕ
    for each $v_i \in V$ **do**
        $\phi \leftarrow \phi \wedge (r_i \vee b_i \vee g_i)$
    for each $e = (v_i, v_j) \in E$ **do**
        $\phi \leftarrow \phi \wedge (\overline{r_i} \vee \overline{r_j}) \wedge (\overline{b_i} \vee \overline{b_j}) \wedge (\overline{g_i} \vee \overline{g_j})$
    return ϕ

By inspection, this is efficient because we are adding a constant number of clauses for every vertex and for every edge. Intuitively, the formula encodes a 3-coloring of the graph: for each vertex v_i, we enforce that it must be assigned at least one color (red, green, or blue), and for each edge (v_i, v_j), we enforce that the endpoints cannot have the same color. If $G \in$ 3-COLORING, then a valid 3-coloring gives a satisfying assignment for ϕ, and conversely, any satisfying assignment for ϕ corresponds to a valid 3-coloring.

Given an efficient decider D_{SAT} for SAT, design and analyze (both correctness and runtime) an efficient algorithm that given $G = (V, E)$, finds an actual 3-coloring for G. The coloring should be a set of vertex-color pairs. If the graph is not 3-colorable, output $\emptyset$.

Solution:

1. On input $G = (V, E)$, construct ϕ via the given reduction: include clauses forcing each vertex v_i to have at least one color $(r_i \vee b_i \vee g_i)$ and clauses ensuring that each edge (v_i, v_j) cannot have the same color $(\overline{r_i} \vee \overline{r_j})$, $(\overline{b_i} \vee \overline{b_j})$, $(\overline{g_i} \vee \overline{g_j})$.
2. Run the efficient decider D_{SAT} on ϕ.
3. If D_{SAT} rejects, output $\emptyset$. Otherwise, obtain a satisfying assignment α from D_{SAT} and construct a 3-coloring:

$$\text{Color } v_i = \begin{cases} \text{red} & \text{if } r_i \text{ is true in } \alpha, \\ \text{blue} & \text{if } b_i \text{ is true in } \alpha, \\ \text{green} & \text{if } g_i \text{ is true in } \alpha. \end{cases}$$

4. Return the set of vertex-color pairs $\{(v_i, \text{Color } v_i)\}$.

Correctness is verifiable in the fact that a valid 3-coloring of G exists if and only if ϕ is satisfiable. Also, constructing ϕ and decoding the assignment both require polynomial time, and calling D_{SAT} is assumed efficient. Thus, the overall algorithm runs in polynomial time.

5. **Approximate f-SET-COVER.**

In this question, you will consider a variant of the SET-COVER problem where each element of the universe is in a limited number of subsets. Formally, in the f-SET-COVER problem, we are given a "universe" (set) U and subsets $S_1, \ldots, S_n \subseteq U$ *where each universe element appears in at most f of the subsets.* The goal is to find a *smallest* number of the subsets that "covers" U, i.e., an $I \subseteq \{1, \ldots, n\}$ such that $\bigcup_{i \in I} S_i = U$. We assume that $\bigcup_{i=1}^{n} S_i = U$, otherwise no solution exists.

You will analyze an "f-cover" approximation algorithm for the f-SET-COVER problem. The algorithm is a generalization of the "double cover" algorithm for VERTEX-COVER from lecture, and it works essentially as follows: while there is some uncovered element u in the universe, add to the cover *all* the subsets that have u in them. The formal pseudocode is as follows. The notation $I(u) = \{i : u \in S_i\}$ for $u \in U$, that is, $I(u)$ indicates which subsets have u.

```
1: function f-COVER(U, S_1, ..., S_n)
2:     I = C = ∅ // selected indices I, covered elements C
3:     while C ≠ U do // not all elements are covered
4:         choose an arbitrary u ∈ U \ C  // element u is not yet covered
5:         I = I ∪ I(u), C = C ∪ ⋃_{i∈I(u)} S_i // add all subsets S_i having u to the cover
6:     return I
```

Fix some arbitrary f-SET-COVER instance, and let I^* denote an optimal set cover for it. Let E denote the set of elements u chosen in Step 4 during an execution of the algorithm, and let I denote the algorithm's final output.

(5 pts) (a) Prove that $I(u) \cap I(u') = \emptyset$ for every distinct $u, u' \in E$. In other words, prove that if u and u' are each selected as the uncovered element in different iterations, then no S_i has both u and u'.

Solution: If some S_i contained distinct $u, u' \in E$, then after adding S_i to cover u, it would cover u' as well, so u' would never be chosen later. Thus, $I(u) \cap I(u') = \emptyset$.

(5 pts) (b) We want a lower bound on OPT $= |I^*|$. Using the previous part, prove that $|E| \leq |I^*|$.

Solution: Each $u \in E$ must be covered by some set in I^*, and no single set can cover two elements from E. Thus, $|E| \leq |I^*|$.

(4 pts) (c) We want an upper bound on ALG $= |I|$. Prove that $|I| \leq f \cdot |E|$, and conclude that the f-cover algorithm is an f-approximation algorithm for the f-SET-COVER problem.

Solution: Each chosen $u \in E$ is in at most f subsets, so each iteration adds at most f sets to I. Thus $|I| \leq f\,|E|$. Combining with $|E| \leq |I^*|$ gives $|I| \leq f\,|I^*|$. Thus, the f-cover algorithm is an f-approximation algorithm for the f-SET-COVER problem.

(6 pts) (d) Prove that for every positive integer f, there is an input for which the f-cover algorithm necessarily outputs a cover that is f times as large as an optimal one.

Solution: For each f, construct an instance with n elements $e_1, \ldots, e_n$. Partition the subsets into n disjoint groups of f each, where each group covers only one distinct element. Each element appears in exactly f subsets. The algorithm picks all f subsets for every newly uncovered element, while an optimal cover picks exactly one subset per element. The algorithm picks a total of fn subsets, while the optimal cover picks a total of n. Thus, the algorithm's cover can be f times as large as an optimal one.

6. **Allocating cashiers.**

You have been appointed as the manager of Meijer, and are responsible for overseeing the team of cashiers. It is closing time, but a number of customers are waiting with full carts.

Your cashiers are tired and ready to go home, so you want to minimize the number of items each cashier needs to scan to get through all of the customers.

We formalize this as the problem of least-item checkout. The input is a number of cashiers n and the number of items $c_1, \ldots, c_m$ that each of the m customers have, i.e., c_i is the number of items that a cashier would need to scan for customer i to check out. The desired output is an allocation of the customers to the cashiers that minimizes the workload. The workload of an allocation is the maximum, over all the cashiers, of the number of items scanned by a single cashier. Each customer must be served by a single cashier, and cashiers process their customers sequentially without breaks.

Formally, this can be defined as:

$$\text{ALLOCATE} = \left\{ (n, C = \{c_1 \ldots c_m\}, k) : C \text{ can be partitioned into } n \text{ subsets where each subset sums to at most } k \right\}.$$

Note: A *partition* of a set S is a collection of subsets $\{S_1, \ldots, S_n\}$ such that $S_i \cap S_j = \emptyset$ for all $i \neq j$ and $\bigcup_i^n S_i = S$.

In this problem, you will first prove that ALLOCATE is NP-complete. Then, you will analyze an approximation algorithm for the minimization version of ALLOCATE.

(6 pts) (a) Prove that ALLOCATE $\in$ NP.

Solution: A witness consists of a partition of C into n subsets such that each subset sums to at most k. Checking that the subsets cover C, have no overlap, and each sum is at most k can be done in polynomial time. Thus ALLOCATE $\in$ NP.

(12 pts) (b) Prove that SUBSET-SUM $\leq_p$ ALLOCATE, where

$$\text{SUBSET-SUM} = \{(A, s) : A \text{ is a multiset of integers} \geq 0, \text{ and } \exists\, I \subseteq A \text{ s.t } \sum_{a \in I} a = s\}.$$

Since SUBSET-SUM is NP-hard (from Homework 8), it follows that ALLOCATE is also NP-hard.

Hint: Consider adding element(s) to A.

Solution: Given $(A, s) \in$ SUBSET-SUM, construct (n, C, k) for ALLOCATE by letting $n = 2$, $k = s$, and $C = A \cup \{x\}$, where

$$x \;=\; \max\{\,0,\; 2s - \textstyle\sum(A)\}\,.$$

Forming C (by adding at most one new element) is polynomial in the size of (A, s).

- ($\Rightarrow$) If there is a subset $I \subseteq A$ summing to s, then put I in one cashier's subset and the rest of $A \cup \{x\}$ in the other. Since $\sum(A) - s + x = 2s - s + x = s$ whenever $x = 2s - \sum(A)$, both subsets have sums at most s. If $2s < \sum(A)$, $x = 0$ and no valid partition exists unless $\sum(A) \leq 2s$, which matches the original instance being "no."
- ($\Leftarrow$) If (n, C, k) is a "yes" instance of ALLOCATE, then at most one subset can contain x. A subset of A must sum to s to fill exactly k when combined with x, or else the other subset would exceed k. So, there is a subset of A with sum s, implying (A, s) is a "yes" instance of SUBSET-SUM.

Thus, SUBSET-SUM $\leq_p$ ALLOCATE, so ALLOCATE is NP-hard.

While this is an NP-complete problem, there is a polynomial-time (greedy) approximation algorithm:

```
1: function GreedyAllocate(n, c_1, ..., c_m)
2:     while there is an unallocated customer do
3:         allocate an arbitrary unallocated customer to a cashier who is allocated the fewest items
4:     return the computed allocation
```

You will analyze this algorithm in the next few parts.

(4 pts) (c) Letting OPT be the optimal workload, prove that OPT $\geq \max\{c_1, c_2, \ldots, c_m\}$. That is, the optimal workload is at least the largest number of items bought by a single customer.

Solution: Since every customer must be allocated to a single cashier, the workload of any allocation is at least the size of the largest set of items held by any single customer. Thus, OPT $\geq \max\{\, c_1, \ldots, c_m\}$.

(5 pts) (d) Prove that if $m \leq n$, GreedyAllocate necessarily returns an optimal allocation.

> **Solution:** If $m \leq n$, allocate each of the m customers to a distinct cashier. Each cashier scans at most $\max\{ c_1, \ldots, c_m \}$ items, which matches the lower bound from part (c). This allocation is necessarily optimal, and the greedy algorithm returns it by placing each customer on a new cashier.

(5 pts) (e) Prove that $\text{OPT} \geq \frac{1}{n} \cdot \sum_{j=1}^{m} c_j$. That is, the optimal workload is at least the *average* number of items scanned by the cashiers.

> **Solution:** Any allocation that handles all items among n cashiers must have at least one cashier scanning at least $\frac{1}{n} \sum_{j=1}^{m} c_j$ items. Thus, $\text{OPT} \geq \frac{1}{n} \sum_{j=1}^{m} c_j$.

(6 pts) (f) Prove that $\text{ALG} \leq 2 \cdot \text{OPT}$, where ALG is the workload obtained by GreedyAllocate.
Hint: Consider a cashier that scans the most items (i.e., their number is equal to the workload), and how the customers were allocated before this cashier's final customer was allocated to them.

> **Solution:** Let ALG be the workload of the final allocation from GreedyAllocate and let i be the last customer assigned to the cashier with workload ALG. Just before adding i, that cashier's total was at most the fewest among all cashiers, so it was at most OPT. Also $c_i \leq \text{OPT}$, since OPT is at least every customer's items. So, $\text{ALG} \leq \text{OPT} + \text{OPT} = 2\,\text{OPT}$.