COMP 3804 — Winter 2026
Solutions Problem Set 7

Question 1: The *set cover problem* is defined as follows:

$$\begin{aligned}\textsc{SetCover} = \{(B, S_1, S_2, \ldots, S_m, K) : \quad & B \text{ is a finite set; } m \text{ is an integer;} \\ & S_1, S_2 \ldots, S_m \text{ are sets with } \cup_{i=1}^{m} S_i = B; \\ & K \text{ is an integer;} \\ & \text{there exists a subset } I \subseteq \{1, 2, \ldots, m\} \text{ of} \\ & \text{size } K, \text{ such that } \cup_{i \in I} S_i = B \ \}.\end{aligned}$$

The *(0-1)-integer programming problem* is defined as follows:

$$\begin{aligned}\textsc{IntProg} = \{(A, K) : \quad & A \text{ is an integer } n \times m \text{ matrix all of whose entries} \\ & \text{are in } \{0, 1\}; \ K \text{ is an integer;} \\ & \text{there exists a binary column vector } x \text{ of length } m \text{ with} \\ & \text{exactly } K \text{ ones, such that } Ax \geq \mathbf{1} \\ & \text{(componentwise) } \},\end{aligned}$$

where $\mathbf{1}$ denotes the column vector of length n, all of whose entries are equal to 1.

Prove that $\textsc{SetCover} \leq_P \textsc{IntProg}$, i.e., in polynomial time, SetCover can be reduced to IntProg.

Solution: We need a function f such that

- f maps $(B, S_1, \ldots, S_m, K)$ to (A, K'),
- $(B, S_1, \ldots, S_m, K) \in \textsc{SetCover} \Leftrightarrow (A, K') \in \textsc{IntProg}$,
- the time to compute (A, K') is polynomial in the length of $(B, S_1, \ldots, S_m, K)$.

Here is how we obtain this function f. Let $n = |B|$. The matrix A has n rows and m columns. The rows are indexed by the elements of B. The i-th column will be the characteristic vector c_i for the set S_i: The binary vector c_i has length n; there is a 1 at position j if and only if the j-th element of B is in S_i. We also let $K' = K$.

Using brute force, we can compute the matrix A in time

$$O\left(\sum_{i=1}^{m} |S_i| \cdot |B|\right),$$

which is polynomial in the length of $(B, S_1, \ldots, S_m, K)$.

Consider a binary column vector x of length m. Let I be the set of indices i such that the i-th component of x is 1. Then Ax is a column vector of length n, which is equal to

$$Ax = \sum_{i \in I} c_i.$$

Note that $\cup_{i \in I} S_i = B$ if and only if every entry in the column vector Ax is at least 1. It follows that the following two conditions are equivalent:

- There exists a subset $I \subseteq \{1, 2, \ldots, m\}$ of size K, such that $\cup_{i \in I} S_i = B$.
- There exists a binary column vector x of length m with exactly K ones, such that $Ax \geq \mathbf{1}$ (componentwise).

Question 2: The *subset sum problem* is defined as follows:

$$\textsc{SubsetSum} = \{(a_1, a_2, \ldots, a_m, b) : \begin{array}{l} m, a_1, a_2, \ldots, a_m \text{ are positive integers,} \\ b \text{ is a non-negative integer, and} \\ \exists I \subseteq \{1, 2, \ldots, m\} \text{ such that } \sum_{i \in I} a_i = b \ \}. \end{array}$$

Assume you have a polynomial-time algorithm A that decides, for any input sequence $(a_1, a_2, \ldots, a_m, b)$, whether or not $(a_1, a_2, \ldots, a_m, b) \in \textsc{SubsetSum}$. Note that this algorithm only returns YES or NO; it does not return anything else.

Design a polynomial-time algorithm B that takes an arbitrary sequence $(a_1, a_2, \ldots, a_m, b)$ as input.

- If $(a_1, a_2, \ldots, a_m, b) \in \textsc{SubsetSum}$, then B returns a subset I of $\{1, 2, \ldots, m\}$ such that $\sum_{i \in I} a_i = b$.
- If $(a_1, a_2, \ldots, a_m, b) \notin \textsc{SubsetSum}$, then B returns NO.

Your algorithm B may use algorithm A as a black box. As always, justify your answer.

Solution: First observe that if $b = 0$, then $(a_1, a_2, \ldots, a_m, b) \in \textsc{SubsetSum}$, because we can take $I = \emptyset$.

Here is the main approach:

- Assume that algorithm A tells us that $(a_1, a_2, \ldots, a_m, b) \in \textsc{SubsetSum}$. Our task is to compute a subset $I \subseteq \{1, 2, \ldots, m\}$ such that $\sum_{i \in I} a_i = b$.
- We "ask" algorithm A if $(a_1, a_2, \ldots, a_{m-1}, b)$ is in $\textsc{SubsetSum}$.
  - If the answer is YES, then we know that $I \subseteq \{1, 2, \ldots, m-1\}$. Thus, we recurse with the input $(a_1, a_2, \ldots, a_{m-1}, b)$.
  - If the answer is NO, then we know that m belongs to I. Thus, we recurse with the input $(a_1, a_2, \ldots, a_{m-1}, b - a_m)$ and remember that m is in the final output I.

Based on this, algorithm B does the following:

- **Algorithm** $B(a_1, \ldots, a_m, b)$:
  - Run $A(a_1, \ldots, a_m, b)$.
  - If A returns NO, then B returns NO.

– If A returns YES, then B runs $C(a_1, \ldots, a_m, b, \emptyset)$, where C is specified below.

Before we give the pseudocode for algorithm C, let us specify what its input and output parameters are:

- **Algorithm** $C(a_1, \ldots, a_k, b', I')$:
  - $(a_1, \ldots, a_m, b)$ is a YES-instance for SUBSETSUM.
  - $(a_1, \ldots, a_k, b')$ is a YES-instance for SUBSETSUM.
  - $I' \subseteq \{k+1, k+2, \ldots, n\}$.
  - This algorithm returns a set $I \subseteq \{1, 2, \ldots, m\}$ such that $I' \subseteq I$ and $\sum_{i \in I} a_i = b$.

Algorithm C does the following:

- **Algorithm** $C(a_1, \ldots, a_k, b', I')$:
  - If $b' = 0$: Return I' and terminate.
  - If $k = 1$: Let $I = I' \cup \{1\}$. Return I and terminate.
  - If $k \geq 2$:
    - Run $A(a_1, \ldots, a_{k-1}, b')$.
    - If A returns YES: Run $C(a_1, \ldots, a_{k-1}, b', I')$.
    - If A returns NO: Run $C(a_1, \ldots, a_{k-1}, b' - a_k, I' \cup \{k\})$.

What is the running time of algorithm B? We assumed that algorithm A has polynomial running time; say $O(m^c)$, where c is some constant. It follows from the pseudocode that algorithm B runs algorithm A at most m times. Therefore, the running time of algorithm B is $O(m^{c+1})$.

Question 3: The *Hamilton cycle problem* is defined as follows:

$$\text{HAMILTONCYCLE} = \{G : \ G \text{ is an undirected graph that has a Hamilton cycle}\}.$$

Let φ be a Boolean formula in the variables $x_1, x_2, \ldots, x_n$. We say that φ is in *conjunctive normal form* (CNF) if it is of the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where each C_i, $1 \leq i \leq m$, is of the following form:

$$C_i = l_1^i \vee l_2^i \vee \ldots \vee l_{k_i}^i.$$

Each l_j^i is a *literal*, which is either a variable or the negation of a variable.

The *satisfiability problem* is defined as follows:

$$\text{SAT} = \{\varphi : \varphi \text{ is in CNF-form and is satisfiable}\}.$$

Prove that HamiltonCycle $\leq_P$ Sat, i.e., in polynomial time, HamiltonCycle can be reduced to Sat.
Hint: Let $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$. Use n^2 Boolean variables x_{ij}, $1 \leq i \leq n$, $1 \leq j \leq n$, where

$$x_{ij} = \mathit{true} \Leftrightarrow \text{ vertex } v_i \text{ is the } j\text{-th vertex in the Hamilton cycle.}$$

Solution: We need a function f such that

- f maps a graph G to a Boolean formula φ in CNF-form,
- G has a Hamilton cycle $\Leftrightarrow$ φ is satisfiable,
- the time to compute φ is polynomial in the length of G.

Let $G = (V, E)$ be an undirected graph with $V = \{v_1, v_2, \ldots, v_n\}$. Recall that a Hamilton cycle is a permutation $u_1, u_2, \ldots, u_n$ of V such that $\{u_1, u_2\}, \{u_2, u_3\}, \ldots, \{u_{n-1}, u_n\}, \{u_n, u_1\}$ are edges in E. We are going to encode the existence of such a cycle as a Boolean formula.

We will use n^2 Boolean variables x_{ij}, $1 \leq i \leq n$, $1 \leq j \leq n$. The meaning of these variables is as follows:

$$x_{ij} = \mathit{true} \Leftrightarrow \text{ vertex } v_i \text{ is at position } j \text{ in the permutation.}$$

Consider a vertex v_i. We note that

$$x_{i1} \vee x_{i2} \vee \ldots \vee x_{in} = \mathit{true} \Leftrightarrow \text{ } v_i \text{ occurs at least once.}$$

For two indices $j \neq k$, $x_{ij} \wedge x_{ik} = \mathit{true}$ if and only if v_i is at positions j and k. Thus, $x_{ij} \wedge x_{ik}$ must be *false*, which is the same as saying that $\neg(x_{ij} \wedge x_{ik})$ must be *true*, which is the same as saying that

$$\neg x_{ij} \vee \neg x_{ik}$$

must be true. Thus,

$$\bigwedge_{j \neq k} (\neg x_{ij} \vee \neg x_{ik}) = \mathit{true} \Leftrightarrow \text{ } v_i \text{ occurs at most once.}$$

Let

$$\varphi_1 = \bigwedge_{i=1}^{n} \left((x_{i1} \vee x_{i2} \vee \ldots \vee x_{in}) \wedge \bigwedge_{j \neq k} (\neg x_{ij} \vee \neg x_{ik}) \right).$$

Then

$$\varphi_1 = \mathit{true} \Leftrightarrow \text{ each } v_i \text{ occurs exactly once.}$$

The size of φ_1 is $O(n^3)$ and it can be constructed in time $O(n^3)$.

We need more to guarantee that we have a permutation of the vertex set. Note that we can make φ_1 *true* by placing all vertices at the same position.

Consider a position j. We note that

$$x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj} = \mathit{true} \Leftrightarrow \text{ there is at least one vertex at position } j.$$

Let

$$\varphi_2 = \bigwedge_{j=1}^{n} (x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj}) .$$

Then

$$\varphi_2 = \mathit{true} \Leftrightarrow \text{ each position has at least one vertex.}$$

The size of φ_2 is $O(n^2)$ and it can be constructed in time $O(n^2)$.

So far, we have that

$$\varphi_1 \wedge \varphi_2 = \mathit{true} \Leftrightarrow \text{ we have a permutation of the vertex set.}$$

Exercise: Convince yourself that we do not need a Boolean formula that expresses "each position has at most one vertex", because it is implied by $\varphi_1 \wedge \varphi_2$ being true.

Let $\{v_i, v_{i'}\}$ *not* be an edge in the edge set of G. We note that

$$(x_{ij} \wedge x_{i',j+1}) \vee (x_{i'j} \wedge x_{i,j+1})$$

must be *false*. Thus,

$$\neg \left((x_{ij} \wedge x_{i',j+1}) \vee (x_{i'j} \wedge x_{i,j+1}) \right)$$

must be *true*, which is the same as

$$(\neg x_{ij} \vee \neg x_{i',j+1}) \wedge (\neg x_{i'j} \vee x_{i,j+1}) .$$

Let

$$\varphi_3 = \bigwedge_{\{v_i, v_{i'}\} \notin E} \bigwedge_{j=1}^{n-1} ((\neg x_{ij} \vee \neg x_{i',j+1}) \wedge (\neg x_{i'j} \vee x_{i,j+1}))$$

and

$$\varphi_4 = \bigwedge_{\{v_i, v_{i'}\} \notin E} ((\neg x_{in} \vee \neg x_{i',1}) \wedge (\neg x_{i'n} \vee x_{i,1})) .$$

Then

$$\varphi_3 \wedge \varphi_4 = \mathit{true} \Leftrightarrow \text{ each neighboring pair of vertices is an edge.}$$

The size of $\varphi_3 \wedge \varphi_4$ is $O(n^3)$ and it can be constructed in time $O(n^3)$.

To conclude, if we let

$$\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4,$$

then φ is satisfiable if and only if the graph G has a Hamilton cycle. The total time to construct φ is $O(n^3)$.

Question 4: Justin Bieber is intrigued by the **P** versus **NP** problem. While taking a shower, Justin suddenly realizes that the problem is actually not very difficult; he comes up with a proof of what is now known as

Bieber's Theorem: P = NP.

Let φ be a Boolean formula in the variables $x_1, x_2, \ldots, x_n$.

We say that φ is in *conjunctive normal form* (CNF) if it is of the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where each C_i, $1 \leq i \leq m$, is of the following form:

$$C_i = l_1^i \vee l_2^i \vee \ldots \vee l_{k_i}^i.$$

Each l_j^i is a *literal*, which is either a variable or the negation of a variable.

We say that φ is in *disjunctive normal form* (DNF) if it is of the form

$$\varphi = C_1 \vee C_2 \vee \ldots \vee C_m,$$

where each C_i, $1 \leq i \leq m$, is of the following form:

$$C_i = l_1^i \wedge l_2^i \wedge \ldots \wedge l_{k_i}^i.$$

Again, each l_j^i is a literal.

We define the following two decision problems:

$$\textsc{Sat} = \{\varphi : \varphi \text{ is in CNF-form and is satisfiable}\}$$

and

$$\text{DNFSAT} = \{\varphi : \varphi \text{ is in DNF-form and is satisfiable}\}.$$

(4.1) Justin starts by proving that DNFSAT $\in$ **P**. Your task is to present a proof of this fact.

Solution: Consider a Boolean formula

$$\varphi = C_1 \vee C_2 \vee \ldots \vee C_m$$

in DNF-form. Thus, each C_i, $1 \leq i \leq m$, is of the form

$$C_i = l_1^i \wedge l_2^i \wedge \ldots \wedge l_{k_i}^i.$$

We need two observations:

- φ is satisfiable if and only if at least one clause C_i is satisfiable.
- The clause C_i is satisfiable if and only if it does not contain a variable, say x_j, and its negation $\neg x_j$.

This leads to the following algorithm:

1. For each $i = 1, 2, \ldots, m$:
   (a) For each $j = 1, 2, \ldots, n$: Check if the clause C_i contains both x_j and $\neg x_j$.
   (b) If there is no such j: return "input is satisfiable" and terminate the algorithm.
2. Return "input is not satisfiable".

The running time of the algorithm is

$$O\left(\sum_{i=1}^{m} n \cdot k_i\right).$$

The size of the Boolean formula φ is something like

$$n + \sum_{i=1}^{m} k_i.$$

Therefore, the running time is at most quadratic in the size of φ.

Note that there are faster algorithms to do this. However, a quadratic running time is enough, because we only care that it is polynomial.

(4.2) Here is Justin's argument to complete the proof of Bieber's Theorem:

- Let φ be an arbitrary Boolean formula in CNF-form. We can use the basic rules of logic (such as De Morgan's Law) to rewrite φ as an equivalent Boolean formula φ' in DNF-form. Therefore,
  $$\textsc{Sat} \leq_P \text{DNFSAT}.$$
- We have seen in **(4.1)** that DNFSAT $\in \mathbf{P}$.
- Since $\textsc{Sat} \leq_P$ DNFSAT and DNFSAT $\in \mathbf{P}$, we have $\textsc{Sat} \in \mathbf{P}$.
- Justin remembers from COMP 3804 that Sat is **NP**-complete.
- Thus, the **NP**-complete problem Sat belongs to **P**.
- Therefore, $\mathbf{P} = \mathbf{NP}$.

Is Justin's proof of Bieber's Theorem correct? As always, justify your answer.

Solution: As you can guess, Justin's proof is wrong. The mistake is in the claim that

$$\textsc{Sat} \leq_P \text{DNFSAT}$$

follows immediately by using the basic rules of logic (that he learned in COMP 1805) to rewrite a CNF-formula as a DNF-formula.

Consider the CNF Boolean formula

$$\varphi = (x_1 \vee y_1 \vee z_1) \wedge (x_2 \vee y_2 \vee z_2) \wedge \cdots \wedge (x_m \vee y_m \vee z_m)$$

where the variables are x_i, y_i, and z_i, for $i = 1, 2, \ldots, m$.

In COMP 1805, you learned that

$$x \vee (y \wedge z)$$

and

$$(x \vee y) \wedge (x \vee z)$$

are logically equivalent. Also

$$x \wedge (y \vee z)$$

and

$$(x \wedge y) \vee (x \wedge z)$$

are logically equivalent.

If we use these rules to convert φ to an equivalent DNF formula, we get a formula with 3^m clauses. Each such clause is of the form

$$c_1 \wedge c_2 \wedge \cdots \wedge c_m,$$

where $c_1 \in \{x_1, y_1, z_1\}$, $c_2 \in \{x_2, y_2, z_2\}$, ..., $c_m \in \{x_m, y_m, z_m\}$. The size of this DNF formula is proportional to 3^m; the time to write it down is thus $\Omega(3^m)$, which is not polynomial in the length of φ.

Sorry Justin! No million dollars for you.