**CS 21 Decidability and Tractability** **Winter 2018**

# Final Exam Solutions

*Posted: March 14*

**If you have not yet turned in the Final you should not consult these solutions.**

1. (a) First, the problem is in PSPACE, for the usual reason for 2-player games. Given $M, B$, we are asking whether $\exists a_1 \forall a_2 \exists a_3 \forall a_4 \cdots Q a_n$ (where each $a_i \in \{1, 2, \cdots, n\}$ and $Q$ is $\exists$ if $n$ is odd and $\forall$ if $n$ is even) such that $M[1, a_1] + M[2, a_2] + M[3, a_3] + \cdots + M[n, a_n] = B$. In particular, we can devise a recursive algorithm (for the slightly more general problem in which the leading quantifier may be either $\exists$ or $\forall$, and $M$ may have fewer than $n$ rows). The algorithm operates as follows: if the leading quantifier is $\exists$, we recursively check (using $n$ recursive calls) whether for there exists $a \in \{1, 2, \ldots, n\}$, for which

$$\forall a_2 \exists a_3 \forall a_4 \cdots Q a_n \sum_{i=2}^{n} M[i, a_i] = B - M[1, a];$$

if the leading quantifier is $\forall$, we recursively check (again using $n$ recursive calls) whether for all $a = 1, 2, \ldots, n$,

$$\exists a_2 \exists a_3 \forall a_4 \cdots Q a_n \sum_{i=2}^{n} M[i, a_i] = B - M[1, a].$$

The base case just involves comparing two integers. This leads to a recursive algorithm with recursion depth $n$ and $\text{poly}(|\langle M, B \rangle|)$ bits of state at each level, for a total space usage of $\text{poly}(|\langle M, B \rangle|)$.

(b) We reduce from QSAT and we refer to the reduction for SUBSET SUM from Lecture 22. An instance of QSAT is a 3-CNF formula $\phi(x_1, x_2, \ldots, x_n)$, with $m$ clauses, and we are asking whether

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots Q x_n \phi(x_1, x_2, \ldots, x_n) = 1.$$

We assume without loss of generality that $n$ is even (we can add a dummy variable if necessary). We now describe the reduction.

In the first row of our matrix $M$ we place two values (repeated as necessary to fill out the whole row), $x_1^{true}$ and $x_1^{false}$ from the SUBSET SUM reduction applied to $\phi$. In the next row we place the two values, $x_2^{true}$ and $x_2^{false}$. We continue in this fashion until we have populated the first $n$ rows of matrix $M$, with the $i$-th row containing the values $x_i^{true}$ and $x_i^{false}$.

The next $2m$ rows are: a row with the three values 0, $FILL1_1$ and twice $FILL1_1$ (again referring to the SUBSET SUM reduction), and a row consisting of all zeros, a row with the three values 0, $FILL1_2$ and twice $FILL1_2$, and a row with all zeros, and so on up to a row with the three values 0, $FILL1_m$ and twice $FILL1_m$ followed by a row with all zeros. The "target" value $B$ we produce in the reduction is the same one produced in the SUBSET SUM reduction applied to $\phi$.

Clearly this reduction runs in polynomial time. We argue that YES maps to YES. In the two-player game interpretation of QSAT (in which the players alternately assign truth values to the variables $x_1, x_2, \ldots, x_n$ with player 1 trying to end with a satisfying assignment), a YES instance $\phi$ implies that there is a win for player 1. In our matrix $M$, the first $n$ rows exactly correspond to the players alternately choosing truth values for $x_1, x_2, \ldots, x_n$, and so there is a strategy for player one that ends this part of the game with a satisfying truth assignment selected. Now, in the remaining $2m$ rows, notice that player 1 is able to pick the required "filler" values (a 0, 1, or 2 in the required digit) for

each of the $m$ clauses, one at a time, to make the sum exactly $B$ (player 2 always gets rows of all zeros, so he can't influence the sum in this phase). Thus there is a win for player 1.

Now we argue that NO maps to NO. As before in the matrix $M$, after the first $n$ rows, the players have alternately selected truth values for $x_1, x_2, \ldots, x_n$, and since $\phi$ is a NO instance, player 2 will be able to ensure that the resulting assignment is not a satisfying one. In particular there will be some clause whose corresponding digit has 0 in the sum so far (since all of its literals are false under the selected assignment), which means that no matter what values player 1 selects in the second phase the sum of $B$ will not be reached. Thus there is not a win for player 1.

2. The language is context free but not regular.

   First we argue that it is context free by describing a NPDA deciding it. The NPDA operates as follows: first it pushes a \$ onto the stack to mark the bottom of the stack, as usual. The idea is for the stack to contain *either* $k \geq 0$ $a$'s which indicate that at this point in processing the string, $k$ more $a$'s than $b$'s have been encountered, *or* $k \geq 0$ $b$'s which indicate that at this point in processing the string, $k$ more $b$'s than $a$'s have been encountered. To accomplish this, we have the following transitions, each of which is a self-loop to single "main" state:

   - when reading an $a$, with $b$ at the top of the stack, pop the $b$
   - when reading an $a$, with \$ or $a$ at the top of the stack, push an $a$
   - when reading a $b$, with $a$ at the top of the stack, pop the $a$
   - when reading a $b$, with \$ or $b$ at the top of the stack, push a $b$
   - when reading a $c$, do nothing to the stack.

   It is clear that each of these rules maintains a stack satisfying the invariant above. Finally, we have rules for when the end of the string is reached. These allow reading an $\epsilon$ with an $a$ or a $b$ at the top of the stack, and transitioning to an "accept" state (i.e., we can accept if we reach the end of the string and there is a non-zero excess of $a$'s over $b$'s, or a non-zero excess of $b$'s over $a$'s). No other transitions to the "accept" state are available so that in all other situations (namely if there is a \$ at the top of the stack), it is impossible to reach the accept state after processing the entire input string.

   Now we prove $L$ is not regular, using the pumping lemma for regular languages. Assume for the purpose of contradiction that $L$ is regular. Let $p$ be the pumping length, and consider the string

   $$w = a^p b^{p+p!}.$$

   Consider any way of writing $w$ as $xyz$ with $|y| > 0$ and $|xy| \leq p$. The second condition implies that $y$ must contain only $a$'s. Let $k = |y|$. We have

   $$xy^i z = a^{p-i} a^{ki} b^{p+p!}$$

   Since $1 \leq k \leq p$, it divides $p!$. So we can choose $i = p!/k + 1$ to obtain the string $a^{p+p!} b^{p+p!}$. This is not in the language, a contradiction. So $L$ cannot have been regular.

3. (a) Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA deciding language $L$. Let $k$ be the length of $y = y_1 y_2 \dots y_k$. We construct a NFA $B$ deciding $L_{-y}$. Machine $B$ will consist of $k+1$ copies of $A$. The first and last copies will have all the transitions of $A$; the others will have no transitions within the states in that copy. $B$'s start state will be the start state of the first copy of $A$, and its accept states will be the accept states of the $k$-th copy of $A$. For every transition in $A$ from state $p$ to state $q$ labeled with $y_1$, we add an $\epsilon$-transition from state $p$ in the first copy of $A$ to the copy of state $q$ in the second copy of $A$. In general, for every transition in the $A$ from state $p$ to state $q$ labeled with $y_i$, we add an $\epsilon$-transition from state $p$ (in copy $i$) to state $q$ (in copy $i+1$).

   For a string $xy_1y_2 \dots y_kz \in L$, we can follow the arcs the machine $A$ would have followed while reading $x$ in the first copy of $A$, then follow the newly available $\epsilon$-transition to the second copy of $A$ (placing us in a copy of the state we would have been in after reading $y_1$), then the newly available $\epsilon$-transition to the third copy of $A$ (placing us in a copy of the state we would have been in after further reading $y_2$), and so on. Finally we arrive at the $(k+1)$-st copy of $A$, in the state we would have been in after reading $y_1y_2 \dots y_k$. Now we proceed in this copy, reading $z$, which leads to an accept state, since $xy_1y_2 \dots y_kz \in L$.

   If a string $w$ is accepted by machine $B$, then there must be a computation path from the start state in the first copy of $A$ to an accept state in the final copy of $A$. Let $x$ be the portion of the string read before departing the first copy of $A$, and let $z$ be the portion of the string read after entering the last copy of $A$. Then by construction $xy_1y_2 \dots y_kz$ must have been accepted by $A$, which completes the proof that $B$ satisfies the requirements.

   (b) Let $M$ be a Turing Machine that recognizes language $L$. We describe a Turing Machine $N$ that recognizes $L_{-y}$. The input to $N$ is a string $w$ of length $n$, and we must accept iff there is some way to "insert" $y$ into $w$ obtaining a string in $L$. More precisely, we must accept iff there exists $x, z$ for which $xz = w$ and $xyz \in L$. There are $n+1$ possible ways to split $w$ into two strings $x, z$ and insert $y$, and we simulate $M$ *in parallel* on each such string. Specifically, for $i = 0, 1, \dots, n$, we define the string $s_i = w_1, w_2, \dots, w_i y w_{i+1}, w_{i+2}, \dots, w_n$, and we simulate $M$ on the various $s_i$ in parallel – meaning that we simulate 1 step of $M$ on each $s_i$, then the second step of $M$ on each $s_i$, then the third step, etc... We halt and accept if any of the simulations accepts.

4. False. By the Time Hierarchy Theorem there are languages decidable in time $O((2n)^{3k}) = O(n^{3k})$ that cannot be decided in time $O(n^k)$. Such a language is in $P$, and is an example of a language in $NP$ that cannot be decided in time $O(n^k)$ regardless of whether or not there is an NP-complete language that can be decided in time $O(n^k)$.

   It is correct reasoning that the "problem" with the assertion is that reductions can run in time much larger than $O(n^k)$, but this is not a *proof* that there is a language in NP that is not decidable in time $O(n^k)$. Deciding a language by applying a reduction to an NP-complete language and the solving the NP-complete language is only one way of deciding it. To really prove that there is no $O(n^k)$ algorithm, one needs the Time Hierarchy Theorem (or an equivalent diagonalization argument).

5. (a) This problem is NP-complete. It is in NP for the usual reasons (given a subset of vertices, we can check in polynomial time whether it has size at least $k$ and whether it

is an independent set in $G$). To show it is NP-hard, we reduce from (3,3)-SAT (from Problem Set 5). We use the reduction from 3-SAT from Lecture 19, but since our starting CNF formula has no variable occurring more than 3 times, the degree of the resulting graph will be at most 4. This is because after placing triangles for each of clauses (or a pair of parallel edges for clauses with two literals and a single node for clauses with a single literal), every vertex has degree at most 2. Then considering each variable $x_i$, its at most three occurrences may require up to two additional edges incident to a node labeled with $x_i$ or $\neg x_i$, in the part of the reduction where we add edges between pairs of contradictory literals. The reduction is the same as before, so it runs in polynomial time, and it remains true that YES maps to YES and NO maps to NO.

(b) This problem is in P. Notice that a graph with maximum degree at most 2 is just a disjoint collection of paths, cycles, and isolated vertices. Thus there is a Hamilton path from $s$ to $t$ iff all of the nodes lie on the (unique) path from $s$ to $t$. Thus to solve the problem in P, we can just perform a DFS from node $s$, see if we reach $t$ and keep a count of how many nodes we encounter, accepting iff we encounter all of the nodes and reach $t$.

(c) HITTING SET-2 is NP-complete. It is clearly in NP because given a purported hitting set $H \subseteq U$ it is easy to check in polynomial time that $|H| \leq k$ and that each $S_i$ has non-empty intersection with $H$.

To show it is NP-hard, we reduce from VERTEX COVER. Given an instance of vertex cover $\langle G = (V, E), k\rangle$, we produce the following instance of HITTING SET-2: $U$ is the set of vertices $V$, and we have a set $S_{(u,v)} = \{u, v\}$ in our collection $\mathcal{C}$ for each edge $(u, v) \in E$. Note that as required, all of the sets in $\mathcal{C}$ have size at most two. We set the bound $k$ in the instance of HITTING SET-2 to be the same $k$ as in the instance of vertex cover.

Now, suppose there is a vertex cover $V' \subseteq V$ of size at most $k$. Then we claim that $H = V'$ is a hitting set, since for each set $S_{(u,v)} \in \mathcal{C}$, one or both of $u, v$ must be in $V'$ and hence in $H$.

In the other direction, suppose there is a hitting set $H \subseteq U$ of size at most $k$. By definition, for each edge $(u, v)$, $H$ must include one or both of $u, v$ since it hits set $S_{(u,v)} \in \mathcal{C}$. Thus $V' = H$ is a vertex cover of size at most $k$.

We conclude that HITTING SET-2 is NP-complete.

(d) This problem is in P. For each way of selecting 750 or more of the first 1000 clauses (which is a large constant number), we produce the 2-SAT instance with those clauses together with all of the other clauses. We can solve this 2-SAT instance in polynomial time, since 2-SAT is in P. If any of these at most $2^{1000}$ instances is a YES instance, then our instance is a YES instance; if all are NO instances, then our instance is a NO instance. The overall running time is at most $2^{1000}$ times a polynomial, which remains a polynomial.

(e) This problem is NP-complete. It is in NP for the usual reason: given a truth assignment, we can check in polynomial time exactly which clauses it satisfies (and then check that all of the first 1000 clauses are satisfied together with at least 3/4 of the other clauses). The reduction is from MAX-2-SAT. Given an instance $(\phi, k)$ of MAX-2-SAT we add 1000 trivially satisfiable clauses $(x_i \vee \neg x_i)$ on fresh variables, and we make these the first 1000

clauses. We note that the reduction from Problem Set 5 showing that MAX-2-SAT is NP-complete produces instances with $10m$ clauses and $k = 7m$, for an integer $m$. If we add $t$ additional trivially satisfiable clauses on fresh variables, then we have (among the clauses other than the first 1000) a total of $10m + t$ clauses, of which $7m + t$ can be simultaneously satisfied iff the MAX-2-SAT instance was a YES instance. Thus choosing $t = 2m$, we find that $9m = (3/4) \cdot 12m$ out of $12m$ clauses are simultaneously satisfiable iff the MAX-2-SAT instance was a YES instance.