CS 21 Decidability and Tractability **Winter 2014**

Final Exam Solutions

Posted: March 19

If you have not turned in the final, obviously you should not consult these solutions.

1. (a) L is in PSPACE via the following recursive algorithm. In general we are given

$$G, k, (S_i, T_i), \ldots (S_n, T_n)$$

and the question is whether player 1 can win (if i is odd) and whether player 2 can defeat player 1 (if i is even).

We produce G' by deleting S_i and G'' by deleting T_i. If i is odd we want to know if player 1 can win so we recursively check whether player 2 can defeat player 1 given each of the following 2 inputs:

$$G', k, (S_{i+1}, T_{i+1}), \ldots (S_n, T_n)$$

and

$$G'', k, (S_{i+1}, T_{i+1}), \ldots (S_n, T_n).$$

If yes to both, then player 1 can't win and we return "no". Otherwise we return "yes". If i is even we want to know if player 2 can defeat player 1 so we recursively check whether player 1 can win given each of the following 2 inputs:

$$G', k, (S_{i+1}, T_{i+1}), \ldots (S_n, T_n)$$

and

$$G'', k, (S_{i+1}, T_{i+1}), \ldots (S_n, T_n).$$

If yes to both, then player 1 can win no matter what player 2 does so we return "no"; otherwise we return "yes".

The base case is checking whether a graph has a clique of size at least k, which is in NP and hence in PSPACE. The overall recursive algorithm thus has polynomial recursion depth and uses polynomial space at each level, so it is runs using only polynomial space.

(b) We reduce from QSAT, as suggested. We have a triple of nodes for each of the m clauses, labeled with the literals in that clause. We add all edges between different triples (but none between nodes of a given triple). Let S_i be the subset of nodes labeled with literals $\neg x_i$ and let T_i be the subset of nodes labeled with literals x_i. Our instance is the graph G just described, the bound m, and the list of n pairs of subsets, one for each variable in the QSAT instance.

Notice that the effect of deleting S_i is thus to remove the literals made false by the assignment x_i =TRUE and the effect of deleting T_i is to remove the literals made false

by the assignment x_i =FALSE. Therefore, in the n rounds of play, the two players alternately select a truth assignment, and the remaining graph has exactly those nodes corresponding to literals that are made TRUE by that truth assignment.

We argue that YES maps to YES. If the QSAT instance is a positive instance, then we know that Player 1 has a strategy so that no matter what player 2 does, the n rounds of play end with the 3-CNF being satisfied. Applying this strategy to the graph game, player 1 can cause the game to end with at least one vertex from each triple, and since we had all possible edges between different triples, this graph contains a m-clique, so the graph game instance we produced was a positive instance.

Now, we argue that NO maps to NO. If the QSAT instance is a negative instance, then we know that Player 2 has a strategy so that no matter what player 1 does, the n rounds of play end with the 3-CNF being *not* satisfied. Applying this strategy to the graph game, player 2 can cause the game to end with at least one triple being entirely deleted (as all of the corresponding literals are FALSE in the assignment the players alternately select). Since the graph only contains edges between different triples, there can be no m-clique in this remaining subgraph (note that a clique can contain at most one node from each triple since there are no edges between nodes in the same triple). Thus the graph game instance we produced was a negative instance.

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 ϵ 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). 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. Language L is neither R.E. nor co-R.E. Reduction from EQ_{TM} (Lecture 14), which is neither R.E. nor co-R.E. Given instance $\langle N_1, N_2 \rangle$ we set $M_1 = M_3 = N_1$ and $M_2 = N_2$. If $L(N_1) = L(N_2)$ then $L(M_1) = L(M_2) = L(M_3)$ and clearly the instance is a positive instance. Otherwise either N_1 is properly contained in N_2 (in which case $L(M_2)$ is not contained in $L(M_3)$, or N_2 is properly contained in N_1 (in which case $L(M_1)$ is not contained in $L(M_2)$.

   (a) Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA deciding language L. Let k be the length of $y = y_1 y_2 \ldots y_k$. 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 ϵ-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 ϵ-transition from state p (in copy i) to state q (in copy $i+1$).

   For a string $xy_1y_2 \ldots y_k z \in L$, 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 ϵ-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 ϵ-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 $y_1y_2 \ldots y_k$. Now we proceed in this copy, reading z, which leads to an accept state, since $xy_1y_2 \ldots y_k z \in L$.

   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 $xy_1y_2 \ldots y_k z$ 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 $i = 0, 1, \ldots, n$, we define the string $s_i = w_1, w_2, \ldots, w_i y w_{i+1}, w_{i+2}, \ldots, w_n$, 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. (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 ϵ-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 ϵ-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 ϵ-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 ϵ-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.

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 Problem Set 5, 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 100 cannot have any

clique on more than 101 nodes. Therefore, we can find the maximum clique size in time n^{101} by enumerating all candidate cliques of up to 101 nodes, and checking each one. We then accept iff k is at most this maximum clique size.

(c) This problem is NP-complete. It is in NP for the usual reasons: given a schedule (all of the s_i can be assumed to be less than P, so writing them down takes only polynomially many bits in the size of the input), it is easy to verify that it is valid, and to sum up the various p_i and compare with P.

To see that it is NP-hard, we reduce from SUBSET SUM. Let $(a_1, \ldots, a_n, B)$ be an instance of SUBSET SUM. Our reduction produces the following instance of the scheduling problem $(a_1, B, a_1), \ldots, (a_n, B, a_n), B$. Now, YES maps to YES: suppose there is a subset I for which $\sum_{i \in I} a_i = B$. Order the elements of I arbitrarily: $i_1, \ldots, i_k$. The schedule that sets $s_1 = 0$, $s_2 = a_{i_1}$, $s_3 = a_{i_1} + a_{i_2}$, $s_4 = a_{i_1} + a_{i_2} + a_{i_3}$, etc... and the remaining s_i arbitrarily (but non-overlapping) results in a profit of exactly $\sum_{i \in I} a_i = B$, as required.

For NO maps to NO: suppose there is a schedule $s_1, s_2, \ldots, s_n$. Note that the potential profit for each job is equal to its duration, and jobs cannot overlap, so because the deadline for all jobs is B, the profit cannot exceed B. Thus, if it is at least B, it must be exactly B. But this means that the sum of those a_i for which job i completes before the deadline is exactly B; thus the instance of SUBSET SUM must have been a positive instance.

(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 (ϕ, 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.