CS 21 Decidability and Tractability **Winter 2024**

Problem Set 7

Out: February 28 *Due:* **March 6**

Reminder: you are encouraged to work in groups of two or three; however you must turn in your own write-up and note with whom you worked. You may consult the course notes and the text (Sipser). The full honor code guidelines and collaboration policy can be found in the course syllabus.

Please attempt all problems. Please select *one* among Problem 1 and Problem 3 to be graded completely (and **indicate clearly** which one); the other one will receive 1 point for a credible attempt. **Please turn in your solutions via Gradescope, by 1pm on the due date.**

1. A function $f : \Sigma^* \to \Sigma^*$ is called *length preserving* if for all $x \in \Sigma^*$,

$$|f(x)| = |x|.$$

A length-preserving function f is called a *permutation* if it is onto, and note that in this case the inverse f^{-1} is a well-defined function. Informally, such an f is *one-way* if it can be computed in polynomial time, but it is hard to invert. Much of cryptography is based on the assumption that there exist *length-preserving, one-way permutations*; indeed the multiplication of two primes (whose inverse is factoring) can be expressed in this way.

Let $f : \Sigma^* \to \Sigma^*$ be a *length-preserving, one-way permutation*, and let $g : \Sigma^* \to \{0, 1\}$ be a polynomial-time computable predicate. A typical scenario is this: you hold a secret x, and you reveal $y = f(x)$ (which is easy to compute). If an adversary can obtain x from y, she is able to compute a bit $g(x)$ that she is not supposed to know. In this problem you will show that the adversary's task is in NP $\cap$ coNP; in other words, you should show that the following language is in NP $\cap$ coNP:

$$L = \left\{ y : g(f^{-1}(y)) = 1 \right\}.$$

A side comment: this shows that it is possible (if P = NP $\cap$ coNP) for the adversary's task to become easy, and much of cryptography to become insecure, while it might still be the case that P $\neq$ NP.

2. In this problem we consider a model of distributed computation. We have n *processes*, modeled as directed graphs $G_1 = (V_1, E_1), \dots, G_n = (V_n, E_n)$. The nodes of graph G_i are the states of process i; it can potentially move from state v to state w iff there is an edge $(v, w) \in E_i$. Here is how we model the *distributed* nature of the computation performed by these n processes: we are given a list of pairs of edges $P = ((e_1, e_1'), (e_2, e_2'), \dots, (e_m, e_m'))$ such that for each i, e_i and e_i' belong to the edge sets of two different graphs from among $G_1, \dots, G_n$. These are intended to specify the possible communications between processes: if process i is in state $v \in V_i$ and process j is in state $v' \in V_j$, then (with some communication), they can move into state w and w' respectively iff the pair $((v, w), (v', w'))$ is in the list P.

Thus the state of the overall system is specified by a vertex in each of the n graphs; i.e., it is an element of $V = V_1 \times V_2 \times \cdots \times V_n$. The relation $T \subseteq V \times V$ (which is determined by $G_1, \ldots, G_n$ and P) describes all possible transitions from one overall state to another. T consists of all pairs $((v_1, v_2, \ldots, v_n), (v_1', v_2', \ldots, v_n'))$ for which there are two indices i, j for which $((v_i, v_i'), (v_j, v_j')) \in P$ and for every other index k, $v_k = v_k'$. In other words, at each step, one of the communications specified in list P may occur, changing the state of the associated two processes, while the other processes remain unchanged.

A central problem in designing distributed systems is avoiding *deadlock*. In this model, a *deadlock state* is a state $v \in V$ for which there is no $w \in V$ with $(v, w) \in T$.

(a) It would be nice to be able to automatically examine a system modelled in this way and determine if there is a deadlock state. Unfortunately this is likely to be hard. Show that the following language is NP-complete:

$$\text{DEADLOCK} = \{(G_1, G_2, \ldots, G_n, P) : \text{directed graphs } G_1, G_2, \ldots, G_n \text{ and the list } P \text{ specify a system with a deadlock state.}\}$$

Hint: reduce from 3-SAT, producing one process for each clause.

(b) In certain scenarios, it is difficult to design a system with no deadlock state. Instead, we aim to design a system in which a deadlock state is not *reachable* from an initial state of the system. As before it would be nice to be able to automatically verify this property. Unfortunately this is likely to be even harder. Show that the following language is PSPACE-hard (it is in fact PSPACE-complete, but in this problem you are not required to show that it is in PSPACE):

$$\text{REACHABLE DEADLOCK} = \{(G_1, G_2, \ldots, G_n, P, v) : \text{directed graphs } G_1, G_2, \ldots, G_n \text{ and the list } P \text{ specify a system with a deadlock state that is reachable from state } v \in V\}$$

Hint: refer to a tableau, in which each cell is either an alphabet symbol x, or a pair (x, q) indicating that the machine is reading this cell, which contains symbol x, while in state q. The number of processes will be the width of the tableau.

3. In class we defined what it meant for a *language* to be decidable in space $t(n)$. To properly define what it means for a *function* to be computable in space $t(n)$, we need to use multitape Turing Machines. We treat tape 1 as a *read-only* input tape, tape 3 as a *write-only* output tape, tape 2 as a read/write work tape, and we only count the space used on tape 2.

Formally, a function $f : \Sigma^* \to \Sigma^*$ is computable in space $t(n)$ if there exists a 3-tape Turing Machine M with the following property: for every $w \in \Sigma^*$, if M is started with w written on tape 1, then M halts with $f(w)$ written on tape 3, while touching at most $t(|w|)$ cells on tape 2, never writing to tape 1, and never reading from tape 3.

Notice that it is possible to perform non-trivial computations while using sub-linear space. In fact, many important functions can be computed using only $O(\log n)$ space.

When we are discussing function computable in polynomial time, we often implicitly use the (simple) fact that if f and g are computable in polynomial time, then their composition is computable in polynomial time. The analogous fact for space-bounded computation is not so obvious, but it is true nonetheless. Suppose functions $f : \Sigma^* \to \Sigma^*$ and $g : \Sigma^* \to \Sigma^*$ are both computable in space $O(\log n)$. Show that the function $h : \Sigma^* \to \Sigma^*$ defined by $h(w) = f(g(w))$ is computable in $O(\log n)$ space.