Computability and Computational Complexity, A.Y. 2024–2025

Guide to the answers

Monday, February 17, 2025

Exercise 1

The following example appears in the Clay Institute webpage to motivate the inclusion the **P** vs. **NP** issue among its Millennium Prize Problems:

> *Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice.*

We are interested in the general problem with N students and n dormitory places (in the example, $N = 400$ and $n = 100$); let the question be "Will you be able to fill all n dormitory places?"

1.1) Prove that the problem is in **NP**.

1.2) Knowing that 3SAT is complete, prove that the Clay Institute problem is **NP**-complete by an appropriate reduction.

Hint — *For point 1.2, observe that the Clay Institute example problem is just the rephrasing of a well known **NP**-complete problem, and imitate the reduction seen in class.*

Solution 1

1.1) Given N students and n dormitory places, the instance size is given by the representation of the two numbers plus the size of the Dean's list of incompatible student pairs. The latter might be as large as the set of all pairs of students, therefore it is quadratic wrt N:

$$O(\log N + \log n + N^2) = O(N^2).$$

The list size clearly dominates the other items.
If a coworker has found a suitable arrangement, they can easily prove it to us by sending us the list of students to be accommodated in the dormitory. This list may come in two forms: either an array of n IDs identifying the students, each having size $O(\log N)$, therefore having total size $O(n \log N)$, or a binary array with N entries, each entry telling us if the student is admitted to the dormitory, total size $O(N)$. In both cases, the list has polynomial size with respect to the instance. To verify that the list is indeed a solution to the problem, we need to take the following steps:

1. check that the list contains n students (a linear scan is sufficient for this);
2. check that no student is listed more than once (quadratic double scan, not needed if we are given the binary array);
3. for every pair of entries, check that the pair does not appear in the Dean's list (double scan of the accommodations list, and for each pair perform a scan of the Dean's list).

The third check is the most complex, but it is clearly polynomial. Therefore, the existence of a solution can be proved with a polynomially-sized certificate that can be checked in polynomial time, which means that the problem satisfies the condition to be in **NP**.

1.2) The problem is INDEPENDENT SET in disguise: the Dean's list is the graph, provided as an adjacency list, N is the total number of vertices, we are looking for an independent set of size n (connected vertices are incompatible students, which cannot be both accommodated in the dormitory). Therefore, the required reduction is the one from 3SAT to INDSET: given a 3CNF formula F with n clauses, create a "student" for every term in every clause (therefore, we have $N = 3n$ students); next, put in the Dean's incompatibility list all pairs of students corresponding to terms from the same clause, and add to the same list all pairs of students corresponding to incompatible terms (i.e., pairs of terms that are in the form $x_i, \neg x_i$). At this point formula F will be satisfiable if and only if it is possible to select n mutually compatible students, corresponding to true terms in F. See Theorem 20 and Fig. 3.1 in the notes.

Observations

Some student proposed other problems (e.g., k-vertex coloring). This might be OK, but they need to clarify what the colors represents (what are they mapped to, in Theorem Clay Istitute problem?).

Exercise 2

In the following property definitions, a finite alphabet Σ is given; $\mathcal{M}$ spans all DTMs on Σ; $x \in \Sigma^*$ spans all strings; finally, $\mathcal{M}(x)$ represents the computation of $\mathcal{M}$ on input x:

$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} : \forall x\ \mathcal{M}(x) \text{ leaves the initial state } q_0 \text{ at the first step}\}\\
\mathcal{P}_2 &= \{\mathcal{M} : \forall x\ \mathcal{M}(x) \text{ never enters the initial state } q_0 \text{ after the first step}\}\\
\mathcal{P}_3 &= \{\mathcal{M} : \forall x\ \mathcal{M}(x) \text{ changes state at every step}\}.
\end{aligned}$$

2.1) Prove that none of the above properties is semantic.

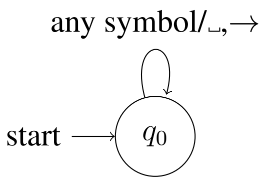
2.2) For each of the properties above, prove whether it is computable or not.

2.3) Prove or disprove the following statement:

Every trivial property of Turing machines is semantic.

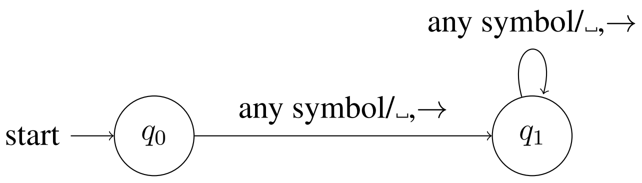
Solution 2

2.1) By definition, a machine that runs forever accepts the empty language. Let $\mathcal{M}_1$ be a machine that keeps going left remaining in the same state:



	0	1	␣
q_0	␣, q_0, →	␣, q_0, →	␣, q_0, →

Let $\mathcal{M}2$ be a machine that in the first step always transition to a new state, then keeps running forever:

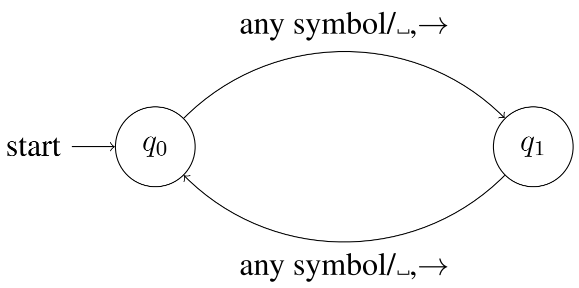


	0	1	␣
q_0	␣, q_1, →	␣, q_1, →	␣, q_1, →
q_1	␣, q_1, →	␣, q_1, →	␣, q_1, →

Clearly, $L(\mathcal{M}_1) = \emptyset = L(\mathcal{M}_2)$, however $\mathcal{M}_1 \notin \mathcal{P}_1$, while $\mathcal{M}_2 \in \mathcal{P}_1$. Therefore, $\mathcal{P}_1$ is not semantic.

The same machines also work for $\mathcal{P}_2$, since $\mathcal{M}_1$ keeps reentering the initial state, while $\mathcal{M}_2$ doesn't.

Observe that neither $\mathcal{M}_1$ nor $\mathcal{M}_2$ change state at every step, therefore neither has property $\mathcal{P}_3$. However, we can build a third machine, $\mathcal{M}_3$, that keeps changing state while running forever:



	0	1	␣
q_0	␣, q_1, →	␣, q_1, →	␣, q_1, →
q_1	␣, q_0, →	␣, q_0, →	␣, q_0, →

Again, $L(\mathcal{M}_3) = \emptyset$, however (unlike $\mathcal{M}_1$ and $\mathcal{M}_2$) $\mathcal{M}_3 \in \mathcal{P}_3$. Therefore, $\mathcal{P}_3$ is not semantic either.

2.2) $\mathcal{P}_1$ is computable: to decide it, we just need to look at the transition function and check that, no matter the input symbol, the rules for q_0 always lead to a different state.

The same idea doesn't work for $\mathcal{P}_2$, since we also need the machine to never go back to q_0; even if there is some rule that leads back to q_0, we need to check if it's ever used. Indeed, the property is non-recursive, and we can reduce the Halting problem (in the version with the empty input, HALT_ε) to it.
Suppose that we are given a TM $\mathcal{M}$, and we want to know if $\mathcal{M}$ ever halts. We can tweak it into a new machine $\mathcal{M}'$ by adding a dummy initial state q_0' that does nothing but immediately move to a new state q_1' that erases the tape (in this way, we ensure that $\mathcal{M}'$ satisfies $\mathcal{P}_2$). Once the tape is erased, $\mathcal{M}'$ transitions to the initial state of $\mathcal{M}$ and starts computing $\mathcal{M}(\varepsilon)$ (because the tape is now empty). However, we replace all transitions to the halting state with transitions to q_0'. Therefore, $\mathcal{M}'(x)$ immediately leaves its initial state, then follows the same steps as $\mathcal{M}(\varepsilon)$; however, if $\mathcal{M}(\varepsilon)$ halts then $\mathcal{M}'(x)$ returns to state q_0', thereby violating property $\mathcal{P}_2$. To summarize, $\mathcal{M}(\varepsilon)$ halts if and only if $\mathcal{M}'(x)$ returns to its original state, which is a different way of saying that $\mathcal{M}(\varepsilon)$ halts if and only if $\mathcal{M}' \not\in \mathcal{P}_2$. We have therefore reduced HALT_ε to $\mathcal{P}_2$, thereby proving that it is uncomputable.

A similar construction lets us prove that $\mathcal{P}_3$ is uncomputable. Again, we reduce HALT_ε to $\mathcal{P}_3$. Let $\mathcal{M}$ be a TM; Let us transform it to a new machine $\mathcal{M}'$ that initially erases it input and then executes the original $\mathcal{M}$ on the empty tape. To make property $\mathcal{P}_3$ valid, we double the number of states of $\mathcal{M}'$ by inserting an "even" and an "odd" version of each state, and making "even" states transition to "odd" ones and vice versa, so that $\mathcal{M}'$ actually changes state at every step, satisfying property $\mathcal{P}_3$. Finally, whenever $\mathcal{M}$ halts, let $\mathcal{M}'$ remain in the same state, thereby violating $\mathcal{P}_3$. Therefore, $\mathcal{M}(x)' \in \mathcal{P}_3$ if and only if $\mathcal{M}(\varepsilon)$ never halts, and we have reduced $\text{HALT}_varepsilon$ to $\mathcal{P}_3$.

Observations

- The fact that the statement of a property mentions states or other syntactic elements doesn't automatically mean that the property is not semantic. Just stick to the definition.
- $\mathcal{P}_3$ does not require the TM to visit a *new* state at each step. If so, the property would be computable, because we just need to check if the machine halts before exhausting all states (and this can be checked for all inputs, because only a finite portion of them can be scanned).
- "To verify the property we need to simulate the machine" is *always* a deeply wrong answer. Simulation is just one out of many possible ways to analyze an algorithm.

Exercise 3

3.1) State Cook-Levin's Theorem.

3.2) Outline the Theorem's proof.

Solution 3

3.1) See the lecture notes.

3.2) See the lecture notes. As an example of correct answer, consider the following outline.

- Let $L \in$ **NP**, and let $\mathcal{N}_L$ be a NDTM that decides $x \in L$ in non-deterministic polynomial time $p(|x|)$.

- We can represent the non-deterministic computation $\mathcal{N}_L(x)$ with a Boolean circuit in $p(|x|)$ stages, where every stage takes the previous configuration and produces the next one. Every stage has polynomial size, because the size of each configuration is polynomial in $|x|$. Every stage takes a bit as an input that represents the non-deterministic choice made by $\mathcal{N}_L$ at that step.

- Therefore, the whole computation is represented as a polynomially-sized Boolean circuit with $p(|x|)$ inputs, and it accepts x if and only if there is a sequence of non-deterministic choices that leads to acceptance.

- The decision $x \in L$ is therefore true if and only if the Boolean circuit has a satisfying input assignment.

- A Boolean circuit can be reduced to a similarly-sized 3-CNF formula. Therefore, the question "$x \in L$?" can be reduced to a 3-CNF formula that is satisfiable if and only if there is an accepting computetion for x.

Many other answers were considered acceptable.