FINAL EXAM 2011 SELECTED SOLUTIONS, CPSC 421/501, FALL 2015

JOEL FRIEDMAN

2

(a) For any integer, $n \geq 0$, Σ^n, the set of strings of length n, is of finite size $|\Sigma|^n$. So we can list all of Σ^* in a sequence by first listing Σ^0, then Σ^1, then Σ^2, etc. Hence Σ^* is countable.

(b) We know that the power set of a set, S, is larger than S, and, in particular, if S is countably infinite then its power set is uncountable. Since Σ is non-empty, Σ^* is infinite (since it has at least one string of each length). Since Σ^* is countably infinite, its power set is uncountable.

(c) If we have a set of programs each of which is described by some string in Σ^*, then the set of programs is countable. If the set of inputs is also Σ^*, then the number of languages is uncountable; hence there are languages that cannot be computed (i.e., decided, recognized, etc.) by any program. [This was an important point in CPSC 421.]

3

See the textbook or class notes for the reduction. Here are the points you need to make to get full credit.

First, you should make diagram like Figure 7.57 on page 321 of the textbook to indicate what the reduction looks like.

Second you should state or indicate the following on your diagram: given a formula, f, with n variables and c clauses, we can build a sequence of $2n + 2c$ integers and a target integer t, such that: (1) exactly one the first 2 integers must be chosen in a subset that sums to the target; it encodes whether the first Boolean variable is true or false; (2) similarly all the first $2n$ integers encode the true/false setting of the n variables; (3) the $2c$ other variables are dummy variables, two per clause to make sure that the digit designated for each clause can sum to 3 iff this clause evaluates to true in the formula.

Third, you should say why this is a polynomial time reduction; namely there are only a polynomial number of integers $2n + 2c$ plus a target, and each is a base 10 integer whose length is $n + c$, and each can be generated by finding the variables

Research supported in part by an NSERC grant.

and seeing which one is in which clause. So each of the integers can be generated in linear time in $\langle f \rangle$, the size of the description of f.

4

On input w to L_1, there a reduction to a string $f(w)$ (an input to L_2) that takes time $p(|w|)$ to generate where p is some polynomial; the length of $f(w)$ is at most $p(|w|)$ (since the tape head can only move one cell per time step).

Similarly there is a polynomial q which is the time that the reduction from L_2 to L_3 requires. It follows that by running the first reduction on w to obtain $f(w)$, and then the second reduction, we take time at most

$$p(|w|) + q(|f(w)|) \le p(|w|) + q(p(|w|)),$$

which is a polynomial in $|w|$. In particular, if $p(n) = n^5$ and $q(n) = n^9$, the time needed on imput w with $|w| = n$ is at most

$$p(n) + q(p(n)) = n^5 + \left(n^5\right)^9 = n^5 + n^{45}.$$

5

DOUBLE-SET is in NP since we can guess two satisfying assignments, which we (1) verify are actually different, and (2) verify that each satisfies the Boolean formula. To produce the guess takes twice with number of variables in the formula, which is less than the description of the formula; each satisfying assignment then takes again polynomial time in the length of the formula.

We now reduce SAT to DOUBLE-SAT. If f is a Boolean formula function, and x_k is a variable not in f, then the formula g defined on one more variable,

$$g = f \wedge (x_k \vee x_k \vee \neg x_k)$$

has twice as many satisfying assignments as f. Thus f has at least one satisfying assignment iff g has two or more satisfying assignments. Given a description of f, we can find a variable x_k that does not appear in f and write down g as above. Clearly this takes polynomial time (we need to find a k such that x_k does not appear in f, and then write down f plus the additional clause). Thus $\mathrm{3SAT} \le \mathrm{DOUBLE-SAT}$.

Hence DOUBLE-SAT is NP-complete.

6

[L_{yes} was called the "acceptance problem" in 2015, and is the analogue of Sipser's A_{TM} in the case of Turing machines.]

Since P is a decider, so is D. Hence the result of D on input EncodeProg(D) is either "yes" or "no." Let us show that either assume leads to a contradiction.

Assume that the result of D on input EncodeProg(D) is "yes;" then the result of P on EncodeBoth(D, EncodeProg(D)) is no; but this means that the result of D on EncodeProg(D) (by definition of P) is "no," which contradicts our assumption.

Similary the "no" assumption leads to the contradiction in the above paragraph with the words "no" and "yes" interchanged everywhere.

Hence L_{yes} is undecidable.

7

Here are some major points to address:

(1) Just by counting, we know that there exist problems that cannot be decided, recognized, etc., since most of the time there are more problems (i.e., languages) than programs. For example, there are only countably infinitely many Turing machines (C programs, Java programs, etc.), but uncountably many problems (i.e., languages). However, counting alone does not tend to produce languages that cannot be decided or recognized (you could generate such a language, but it would involve enumerating all Turing machines and a type of diagonalization algorithm that is probably not of practical interest).

(2) By a type of self-referencing argument, we saw that the acceptance problem and the halting problem are undecidable; furthermore, their complements aren't even recognizable. At times this related to concrete problems, such as the "dead code" problem (i.e., is there a line in a computer program that will never be reached). While compilers can find and eliminate some code that is never used, they can't generally find all of it; so you shouldn't try to build a compiler that tries to find all of it for every possible computer program.

(3) NP-completeness type problems arise much more in practice, as certain optimization problems (some scheduling problems, traveling salesman, etc.) Most people suspect that NP-complete problems cannot be solved in polynomial time; at the very least we don't know algorithms for them. So while such optimization problems can be solved (or approximately) solved) in polynomial time in many practical situations, you should realize that looking for an algorithm that completely solves such problems in polynomial time is an extremely ambitious task.

8 Not covered in 2015.

Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, CANADA, and Department of Mathematics, University of British Columbia, Vancouver, BC V6T 1Z2, CANADA.

E-mail address: jf@cs.ubc.ca or jf@math.ubc.ca

URL: http://www.math.ubc.ca/~jf