Introduction to Theoretical Computer Science

Exercise Sheet: Week 7 – Solutions

(1) The basic claim was that polynomial problems are 'easy', and non-polynomial problems are hard. Consider $f(n) = n^{10^{10}}$, and $g(n) = 10^{n/10^{10}}$. Show that $f(n) \in o(g(n))$. (Recall this means that $\forall \epsilon > 0. \exists n_0. \forall n > n_0. |f(n)| \leq \epsilon |g(n)|$.) (Hint: take logs, and remember that you only have to care about large enough n.) Where does g catch up with f?

Given ϵ, we want to find where $f(n) \leq \epsilon g(n)$, i.e. $n^{10^{10}} \leq \epsilon \cdot 10^{n/10^{10}}$. This is not soluble in closed form, so let's just find n big enough. Taking logs, $10^{10} \log n \leq \log \epsilon + n/10^{10}$. Divide by $\log n$ to get $10^{10} \leq \log \epsilon / \log n + (n/\log n)/10^{10}$. If we take $n \geq 1/\epsilon$, then $-1 \leq \log \epsilon / \log n \leq 0$, so now we just need $(n/\log n) \geq (10^{10}+1) \cdot 10^{10}$, so taking $n/\log n \geq \max(1/\epsilon, 10^{21})$ will certainly do.

The point where $g(n) = f(n)$ is best found numerically, and is about 2.133×10^{21}, pretty much where the above proof put it for $\epsilon = 1$.

Bonus: Where does the statement $f(n) \in o(g(n))$ fit in the arithmetical hierarchy that we discussed unofficially? (Trick question!)

It's a trick question because as presented ϵ is a real, so this is not an arithmetical statement. However, we can replace $\forall \epsilon > 0. \ldots \epsilon \ldots$ by $\forall N > 0. \ldots {}^1\!/\!_N \ldots$ to get an equivalent arithmetical statement, which is in Π^0_3.

(2) We defined the class P in terms of polynomially bounded machines. Explain how to implement this definition. That is, given a register machine M (taking input R in R_0 as usual), explain how to construct a machine M' which takes inputs R and k, and behaves like M except that it halts after $(\lg R)^k$ steps of M's execution.

M' first computes $(\lg R)^k$, by repeated division by 2 (and division by 2 is itself repeated subtraction) and brute force exponentiation (note that we don't care how long this takes since k is a constant), and stashes it in a dedicated register R_{-1} (or whatever). M' now jumps to a program that is the program of M, except that every instruction I is replaced by the sequence DECJZ$(-1,$HALT$); I$*, with labels adjusted appropriately.*

(3) Show that the Halting problem is not NP-complete. (This is obvious ... but can you prove it?)

Suppose it is. Then there is a non-deterministic Turing/Register Machine M and a polynomial time bound $f(n)$ in the size of the input machine such that M determines the halting answer before time $f(n)$. Let k be the maximum branching degree of the control graph of M – this is at most 2 for an NRM, or (number of states) for an NTM. Then we can simulate M deterministically by interleaving, in time $O(k^{f(n)})$, and thus solve the halting problem.

The following is a reasonably tricky algorithm design problem.

(4) 2-SAT is the following problem: given a set of boolean variables X_i, and a formula $\phi = \bigwedge_{1\le j\le n}(\alpha_j \vee \beta_j)$, where each α_j, β_j is a literal, i.e., either a variable or a negated variable, is there a satisfying assignment for ϕ?

Show that 2-SAT is polynomial (unlike SAT or 3-SAT). (Quite difficult. Hint: look for two clauses that contain a variable and its negation (e.g. $(X \vee Y)$ and $(Z \vee \neg Y)$), merge them into a single clause, and add it to the formula.)

A proof by using resolution: *If a variable only occurs positively in ϕ, we may as well set it to true and forget about it (removing any clauses in which it occurs); if only negatively, set it to false, and forget about it (removing it* ***from*** *any clause in which it occurs, leaving just the other disjunct). So we're left with variables that occur both positively and negatively. Clearly, if we now have a clause (Y) and a clause $(\neg Y)$, we can't satisfy ϕ. So suppose we still have two-literal disjuncts, so, for example, $(\alpha \vee Y)$ and $(\beta \vee \neg Y)$. If these are jointly satisfiable, then we can also satisfy $\alpha \vee \beta$, so* resolve *by adding $(\alpha \vee \beta)$ to ϕ (and conversely, if they're not jointly satisfiable, adding the clause does no harm). Repeat until there are no such pairs remaining unresolved. If at any point we end up with a clause (Y) and a clause $(\neg Y)$ for some Y, we can't satisfy; otherwise we can. This takes polynomial time (about n^4).*

An alternative proof via graph theory: *Every clause can be written as an implication $(\alpha \Rightarrow \beta)$ where α, β are literals. Consider a graph where the vertices are literals and directed edges describe these implications. The formula is satisfiable if we never have that $\alpha \Rightarrow \neg\alpha$ for any literal α. This is the case if and only if, in every strongly connected component of the graph a literal does not appear both positively and negatively.*