

NP vs coNP¹

0.1 Introduction

A decision problem is a question that requires a yes or no answer. We wish to consider the computability of such problems in terms of the theoretical bounds on the time taken to produce the answer; in particular we are interested in the possibility of solving problems in polynomial time.

0.2 Some set theory

Definition 0.2.1. We say that a set of problems A is **many-to-one polynomial-time reducible** to another problem set B , denoted $A \leq_p B$ iff there exists a recursive function f st $f(x)$ can be computed in polynomial time and

$$x \in A \Leftrightarrow f(x) \in B$$

Definition 0.2.2. The set A is **hard** for a class of sets L iff every set in L is many-to-one reducible to A .

Definition 0.2.3. The set A is **complete** for a class of sets L iff $A \in L$ and A is hard for L .

0.3 Complexity classes

Definition 0.3.1. The **P complexity class** is the set of problems that can be solved in polynomial time (i.e., solved by a deterministic Turing machine in polynomial time).

Definition 0.3.2. The **NP complexity class** is the set of problems that can be checked in polynomial time (i.e., given a candidate solution, testing the solution is possible in polynomial time). Equivalently, these are the problems which can be solved in polynomial time on a nondeterministic Turing machine.

Definition 0.3.3. In line with definition 0.2.3, the **NPC complexity class** is the set of NP-complete problems. Thus, a problem A will be in NPC iff $A \in NP$ and all other problems in NP can be reduced to A .

Definition 0.3.4. The Satisfiability problem SAT: A boolean formula f is *satisfiable* if there exists a truth assignment T such that $f(T)$ is true.

Clearly, any problem in P is in NP. But it may be that there are problems in NP which are not in P- this will occur if there is no good (i.e., polynomial) way to determine the candidate solutions. An example is possibly given by SAT. Checking a particular truth assignment is clearly in polynomial time, but finding an assignment by the naive method (truth table construction; test all candidates) is exponential in the number of variables, rather than polynomial. Can we do better?

If any problem $X \in NPC$ turns out to be in P , then $Y \in P$ for all $Y \in NPC$; that is, $P = NP$.

¹Ben Barnard, Aili Just, Anna Skeoch, Graeme Taylor, Neil Taylor May 5, 2006

0.4 Cook's Theorem

However, proving directly that an NP problem is NP-hard (i.e., in NPC) is very difficult. Fortunately, to show that an NP problem X is in NPC it suffices to show that X is many-to-one reducible to SAT. That SAT is NP-complete was the first such result (in 1971); using the reduction technique more than ten thousand problems in NPC have been identified.

Theorem 0.4.1 (Cook's Theorem). *SAT is in NPC.*

Sketch Proof: Firstly, the problem is in NP since a non-deterministic Turing machine can simply branch out into the different truth assignments, terminating if any branch is true.

A problem in NP can then be translated into a satisfiability problem using the non-deterministic Turing machine M which accepts it. To do this, construct a function $f(x)$ which is a boolean formula to express whether the machine M accepts x . This can then be run on the Turing machine for the satisfiability problem and thus the original problem reduces to SAT.

Cook's complete proof can be found in [1].

0.5 Co-NP

Definition 0.5.1. **co-NP** is the class of languages L for which the complement L^C is in NP. i.e., it is the class for which there is a non-deterministic Turing machine that accepts a word w iff $w \notin L$.

Definition 0.5.2. Thus, by definition 0.2.3, a problem is **co-NP complete** if it is in co-NP and is co-NP hard.

Definition 0.5.3. The Validity problem VAL: A formula F is *valid* if it is true for all possible truth assignments.

VAL is co-NP complete: F is in VAL iff $\neg F$ is not satisfiable; so the complement of VAL reduces to SAT in polynomial time (since constructing the negation is polynomial). Thus the complement of VAL is in NPC, so VAL is in Co-NPC.

Lemma 0.5.4. $co - P = P$

Proof. Since there is a Turing machine M that accepts a language L in polynomial time, then there is one which accepts L^C in polynomial time: simply run M then reverse the output (give TRUE if M gave FALSE and give FALSE if M gave TRUE). ■

Theorem 0.5.5. *If $NP \neq co-NP$ then $P \neq NP$.*

Proof. If $P = NP$, then (by lemma 0.5.4) $co-NP = co-P = P = NP$. Thus by contraposition, if $NP \neq co-NP$ then it cannot be that $P = NP$. ■

References

- [1] Cook, Stephen *The complexity of theorem-proving procedures* in Proceedings of the Third Annual ACM Symposium on the Theory of Computation 1971 pp.151-158.
- [2] Hopcroft, Motwani and Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison Wesley.