

## Handout 9

### $\mathcal{P}$ versus $\mathcal{NP}$

**1. Solving versus checking.** In the last handout we introduced the fundamental distinction between problems which have a solution (i.e. program) which runs in polynomial time, and problems for which such a solution does not exist. We say that problems of the former kind are **feasible**, while the latter are called **intractable**. We have introduced the notation  $\mathcal{P}$  for the class of all feasible problems.

Of course, the distinction between  $\mathcal{P}$  and not- $\mathcal{P}$ , while straightforward to define, is not one which is easy to establish in practice. When studying a particular problem we might be very interested to know whether it has a feasible solution or not. The only way to show this is by giving an algorithm which runs in polynomial time. Even if we know of one or several algorithms to solve a problem, we may still have a hard time establishing its worst-case behaviour. Similarly, if we suspect that a problem can not be solved in polynomial time, establishing this hypothesis is not easy; we must prove a lower bound for it.

There is a whole range of very common problems for which we have no idea whether they are feasible or not. They are called the  **$\mathcal{NP}$ -complete problems**. This handout tries to explain what this means. In a nutshell, the situation for  $\mathcal{NP}$ -complete problems is as follows: although we do not know whether the problems are really hard, we have very good grounds to believe that they are.

Let us first explain the class  $\mathcal{NP}$ . The easiest way to do so is via the distinction between **solving** a problem and **checking** a solution. It is common experience that the latter is far easier.  $\mathcal{NP}$ , then, consists of all those problems for which there exists a polynomial solution checking algorithm.

**2. Examples.** Here are some examples to illustrate the phenomenon.

**Factorization:** It is very easy to check that one number is divided by another one, no matter how large the numbers involved. We all know how to do “long division”, and it, like the other operations of simple arithmetic, takes time that is linear in the number of digits used. On the other hand, finding a factor of a given number  $n$ , even when it is known that the number is not prime, appears to be extremely hard. An obvious algorithm, to check divisibility for all numbers up to the square root of  $n$ , is exponential in half the number of digits.

**Hamiltonian paths:** Here one is given a graph (directed or undirected) and the problem then is to find a path in the graph that visits each node exactly once. Again, if we are given a solution, it is utterly trivial to check that it satisfies the requirements. Finding such a path, however, seems to be very difficult in general. The Hamilton path problem is one of many  $\mathcal{NP}$ -complete graph problems. Finding efficient algorithms for them would be of immense importance in problems such as chip design or route planning.

**SAT:** This is the problem of satisfiability of propositional expressions. Recall from your Logic course that propositional formulas are generated by the following grammar:

$$\begin{aligned}\phi & ::= A \mid F \mid T \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi \\ A & ::= p \mid q \mid r \mid \dots\end{aligned}$$

Here  $p, q, r$ , etc, are the atomic formulas, or propositional variables. They are placeholders for propositions which are either true or false. For a given assignment of truth values to atomic formulas, the truth value of composite formulas is given by the usual truth tables.

Notation varies for the logical connectives. ‘ $\wedge$ ’ and ‘ $\neg$ ’ stand for ‘and’ and ‘not’. You may be familiar with the alternative symbols ‘ $\&$ ’ and ‘ $\sim$ ’.

The satisfiability problem consists of deciding for a given propositional formula whether there exists an assignment of truth values to atomic formulas such that the given formula becomes true as a whole. For example, the formula

$$\neg(q \rightarrow p) \wedge r$$

is true if we assign false to  $p$  and true to  $r$  and  $q$ . The formula

$$\neg((\neg p \vee q) \rightarrow \neg(p \wedge \neg q))$$

on the other hand, is always false. It is unsatisfiable.

We have to say how a Turing machine would deal with propositional formulas. To this end we assume an input alphabet consisting of  $p, F, T, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, (, ), 0$  and  $1$ . We have only one letter for atomic propositions, so we use  $0$  and  $1$  to generate infinitely many expressions consisting of  $p$  followed by a binary number. Formulas can thus be encoded as strings over this alphabet, and we have an unambiguous definition of the length of a formula.

**Minesweeper consistency:** This example looks frivolous, but I’ve included it because you might enjoy looking at Richard Kaye’s proof that it is  $\mathcal{NP}$ -complete. He is in the School of Maths here at Birmingham, and his website <http://for.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm> gives further details.

I'm not going to describe the minesweeper game. Many of you will be familiar with it because it is included with Windows operating systems. Its consistency problem is the decision problem for whether a given configuration of flags (for conjectured mines) and numbers as in the Minesweeper game can actually arise from a distribution of mines. Checking a mine distribution to see if it matches the configuration is easy, but finding a matching distribution is much harder.

3. **To be precise:** Consider an input alphabet  $\Sigma$  and a relation  $G$  from  $\Sigma^*$  to  $\Sigma^*$  – in other words, a subset of  $\Sigma^* \times \Sigma^*$ . This  $G$  is going to be our starting point for examples such as those just listed. It is the set of pairs  $(s, w)$  for which  $w$  is a satisfactory solution for the problem when  $s$  is the input.

Obviously we can understand  $G$  as a language over the augmented alphabet  $\Sigma \cup \{\#\}$  where  $\# \notin \Sigma$ , by representing a pair  $(s, w)$  as  $s\#w$ . Hence  $G$  has its own decision problem. This is the **checking problem** for  $G$ .

The **solving problem** for  $G$  is, given an input  $s$ , to find an output  $w$  such that  $(s, w) \in G$  – or to tell us that there is no such  $w$ .

If the checking problem is decidable, then the solving problem is semicomputable in the sense that we can always find a solution if one exists, by searching through all possible  $w$ s one by one and checking them. But if there is no solution then the search will never terminate.

We are interested in complexity, and in particular when the checking problem has a polynomial time decider. To get a useful theory, we shall need to restrict our attention to those relations  $G$  where there is a polynomial bound on the lengths needed for solutions: given  $s$ , if there is *any* solution  $w$ , then there is one that is not too long. More precisely, there is some integer  $k$  and some function  $f \in O(n^k)$  such that if  $(s, w) \in G$  then there is some  $(s, w') \in G$  such that  $|w'| \leq f(|s|)$ .<sup>1</sup> We'll say that  $G$  has **polynomial bound on images**. All the examples above have this property.

The first consequence of this is that we can always either compute a solution or say that there are none: for, given  $s$ , we search through the *finitely many*  $w$ s whose length is less than  $f(|s|)$ . If we don't find a  $w$  with  $(s, w) \in G$  then we know that there are none. That much follows just from having the bound on the length of  $w$  – it didn't have to be polynomial.

The second consequence, which *does* depend on it being polynomial, will appear later when we come to non-deterministic Turing machines.

Finally, from  $G$  we get an **existence problem**. Given  $s$ , this is the question of whether there is *some*  $w$  such that  $(s, w) \in G$ . Logically, it is  $(\exists w)(s, w) \in G$  and is a property of  $s$ . This, like the checking problem, is a decision problem.

Throughout we shall be working from some  $G$  that (a) has a polynomial time decider for its checking problem, and (b) has polynomial bound on images. We know already that the existence problem is decidable and the solving problem is computable, by searching through all  $w$ s within the bound on the length. However, that is almost certain to be an exponential algorithm, since the number of  $w$ s with a given length  $m$  is  $|\Sigma|^m$ . Are those problems polynomial?

First, could it be that the existence problem is always polynomial? Nobody knows. That is the " $\mathcal{P} = \mathcal{NP}$ " question. And the solving problem is harder – it answers the existence problem, but in addition provides a solution if one exists.

Keep clear in your mind how, from a relation  $G$ , we get three problems: for checking, solving and the existence problem. Here is an example to show the three.

**Example:** For factorization,  $G$  is the set of  $(x, y)$  such that  $y$  is a factor of  $x$  (other than 1 and  $x$ ). If  $y$  is a factor of  $x$ , then  $y \leq x$ . Hence, when you remove leading 0s from  $y$  its string is no longer than that of  $x$ . Hence this  $G$  has polynomial bound (linear, in fact) on images.

The *checking* problem is easy – given  $x$  and  $y$  you do the division and see if there is any remainder. The *solving* problem is, given  $x$ , to find a factor  $y$ . Algorithms are known that are better than exponential but nobody has found one that is polynomial. The *existence* problem is, given  $x$ , to say (Yes/No) whether it has a factor – or, looking at it the other way round, to say whether  $x$  is prime. In fact, for this  $G$  the existence problem is polynomial. The AKS algorithms based on that of Agrawal, Kayal and Saxena (2002) can test primality in time that is of the order of the 6th power of the number of digits of  $x$ .

The public-key cryptosystem RSA is based on this fact, that the existence problem is feasible but the solving problem seems not to be. Using RSA relies on finding two large primes which are then multiplied together, so a good primality test such as AKS is necessary. The security of RSA relies on it being hard to "unmultiply" those primes, to find them as the factors of their product. No one has proved that there is no polynomial factorization algorithm, but unless some one finds such an algorithm RSA will remain hard to break.<sup>2</sup>

4.  **$\mathcal{NP}$  decision problems.** To keep things simple we shall define  $\mathcal{P}$  and  $\mathcal{NP}$  just for decision problems, i.e. recognition problems for subsets  $L$  of  $\Sigma^*$ . The definition of  $\mathcal{P}$  is easy –  $L$  (or its decision problem) is in  $\mathcal{P}$  if it has a decider that is polynomial time in its input length.

Since there are only two answers "Yes" and "No" ( $s \in L$  or  $s \notin L$ ) for a decision problem, there is little to be gained by thinking about algorithms to check whether either of those is the correct answer. Instead what we do is to think of relations  $G$  for which  $L$  is the existence problem. We need something in addition, which we could call a "proof" or a "witness". In the examples from before these would be the factors of the number (which gives us more information than the simple statement "number is composite"), or an explicit path through the graph (rather than just saying "there exists a Hamiltonian path"). The checker can then verify the answer by multiplying the factors or following the path, respectively. The idea of a witness is formalised as follows:

**Definition 1.** A subset  $L$  of  $\Sigma^*$  belongs to  $\mathcal{NP}$  if there is a relation  $G$  from  $\Sigma^*$  to  $\Sigma^*$  such that

<sup>1</sup>I'm writing  $|s|$  for the length of a string  $s$ .

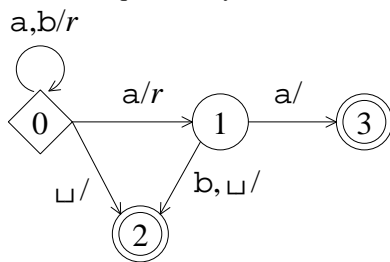
<sup>2</sup>This discussion is based on the "classical" computers that we have at present. In the future we hope to have computers whose algorithms exploit features of quantum physics, and on those *Shor's algorithm* would be able efficiently to factor large numbers and hence break RSA. The present state of the art (2010) is that a quantum computer has factorized 15.

- $L$  is the existence problem for  $G$ ,
- $G$  has a polynomial time decider (a checker for  $L$ ), and
- $G$  has polynomial bound on images.

If  $(s, w) \in G$  then you might think of  $w$  as a “proof” or a “witness” or a “certificate” of  $s$  being in  $L$ .

The checker is making a definite decision about  $w$ : that it is, or is not, a valid witness for  $s$ . But from the point of view of  $L$  there is an asymmetry here. A “Yes” answer says that  $s$  is in  $L$ , but a “No” answer tells us nothing. To say that  $w$  is not a valid witness is not the same as saying  $s$  is not in  $L$ . Therefore we shall call the two possible answers from the checker “Yes” and “Undecided”.

**5. Nondeterministic Turing machines.** The class  $\mathcal{NP}$  can be defined in an entirely different way using **nondeterministic Turing machines** (NTMs). NTMs are defined just like deterministic ones except that we employ a transition *relation* rather than a transition *function*. In a transition table, every entry consists of finitely many choices for how the machine may proceed. We assume that there is at least one possible transition in every case (possibly to an error state). When such a machine is run, it is allowed to choose freely at every stage one particular transition from the available collection. Here is a small example, which is simply a Turing machine form of what could just as well be a finite state machine. (You should be able to see quite easily how to make a deterministic Turing machine out of it.)



This machine, nondeterministically, checks whether there is an occurrence of two consecutive  $a$ 's in a string. It is nondeterministic because, from state 0 reading symbol  $a$ , there are 2 possible transitions, to state 0 or 1.

States 3 and 2 are for replies “yes” and “undecided”. If the machine ever terminates in state 3, then you know the input has two consecutive  $a$ 's.

Conversely, if the input has two consecutive  $a$ 's then it is possible to choose the transitions carefully so as to reach state 3. However, it is also possible to choose them carelessly to reach state 2. Hence termination in state 2 doesn't tell you anything – there may or may not be two consecutive  $a$ 's. This is just like nondeterministic finite state automata used to recognize languages. If a string is in the language then it is possible to choose transitions to reach an accepting state, but other choices may fail to do that.

Note that the relationship between NTMs and deterministic ones is just like that between nondeterministic finite automata and their deterministic counterparts. Also note that here, just like there, we are interested in nondeterminism only as a **descriptive device**. We have no desire to ever build a real machine which exhibits uncontrolled nondeterminism.

**6. NTMs for decision problems.** As before, let us concentrate on decision problems. The NTM must always terminate, but the possible replies are “Yes” ( $s \in L$ ) or “Undecided” (don't know).<sup>3</sup> These must be correct in the sense that a “Yes” reply can happen only if  $s \in L$ . The main definition is this: An NTM is said to **solve** a decision problem if it is correct and if for every input in  $L$  there is *at least one* computation path along which the machine will reply “Yes”.

We can use such an NTM to construct a deterministic machine which truly decides the decision problem (only, it takes a long time to do so). What the deterministic machine does is systematically to explore all execution paths the nondeterministic machine is capable of. Since the NTM is bound to stop after a fixed amount of time, there are only finitely many paths to consider.<sup>4</sup> If one branch ends with outputting “yes”, then the deterministic machine answers “yes” as well. If all branches end in “undecided” the deterministic machine answers “no”.

Note that this simulation does not prove  $\mathcal{P} = \mathcal{NP}$  because the deterministic exploration takes an exponential amount of time in general.

**7. The second definition of  $\mathcal{NP}$ .** The worst-case complexity of a NTM is defined almost as in the deterministic case: We take the maximum runtime (or memory usage) on all input of a certain length, and *for all possible choices the machine can make*. It therefore makes sense to speak of NTMs of polynomial time complexity: Their runtime is bounded by some polynomial independently of the choices they can make during a computation.

**Definition 2.** A decision problem belongs to the complexity class  $\mathcal{NP}$  if there exists a nondeterministic Turing machine running in polynomial time that solves the problem.

Let us argue that the two definitions we have given for  $\mathcal{NP}$  are equivalent. In the one direction, if we have a deterministic machine which checks witnesses, we can construct a non-deterministic one which works on the string under consideration alone, by using an initial phase in which the NTM *guesses* a witness, and a second phase in which the witness is checked as before. In the other direction, when we have an NTM and want to construct a deterministic solution checker what we do

<sup>3</sup>In practice it may be sensible to allow “No” replies ( $s \notin L$ ) too. However, we are not going to require the machines to be able to reply “No” for strings  $s \notin L$ , so we might as well consider only the weak machines that never bother to reply “No”.

<sup>4</sup>The mathematical argument for this in general is rather subtle and is a consequence of what is known as *König's Lemma*. It is easier to see in the case of the polynomial time NTMs that we shall be dealing with.

is to check the computation of the NTM, taking as a witness a string of bits which indicates which choices the NTM has to make in order to reach the “yes” outcome.

The second definition makes it easy to see that  $\mathcal{P}$  is contained in  $\mathcal{NP}$  because deterministic Turing machines are also NTMs. The question whether the two classes are different or not is a famous open question in the Theory of Computation. It is commonly referred to as the

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$$

problem.

**8. Complete problems.** Some problems in  $\mathcal{NP}$ , such as the factorisation problem for natural numbers, have been studied for more than a hundred years, yet up to this day no efficient solution has been found. This strongly suggests that  $\mathcal{NP}$  is a larger class of problems than  $\mathcal{P}$ : so  $\mathcal{P} \neq \mathcal{NP}$ . One would prove this conjecture by selecting some particularly hard problem in  $\mathcal{NP}$  and showing a lower bound for the problem complexity with growth type worse than any polynomial.

But suppose the truth is the opposite, that  $\mathcal{P} = \mathcal{NP}$ . How we would prove that? It looks like a long task of proving that every  $\mathcal{NP}$  problem has a  $\mathcal{P}$  decider. However, it turns out that there are some  $\mathcal{NP}$  problems that are as hard as it gets, and if any one of them has a  $\mathcal{P}$  decider then so do all  $\mathcal{NP}$  problems. These are the  $\mathcal{NP}$ -**complete** problems.

This idea relies on a notion of comparing problems within  $\mathcal{NP}$  by **polytime reducibility**. We say that a problem  $\mathcal{X}$  reduces in polynomial time to problem  $\mathcal{Y}$  if there exists an algorithm (ie, a deterministic Turing machine) which translates instances of  $\mathcal{X}$  to instances of  $\mathcal{Y}$ , and whose runtime is polynomially bounded.

If  $\mathcal{X}$  and  $\mathcal{Y}$  are in  $\mathcal{NP}$  and if  $\mathcal{X}$  can be reduced to  $\mathcal{Y}$  in polynomial time then it is reasonable to say that  $\mathcal{X}$  is “no harder” than  $\mathcal{Y}$ , because any polynomial Turing machine solving  $\mathcal{Y}$  gives rise to a polynomial solution for  $\mathcal{X}$ .

This is the same notion of reducibility as in Handout 5, except that we now also bring time into the picture.

Now, it is true that within  $\mathcal{NP}$  there are some nontrivial reductions between problems, and it could be that  $\mathcal{NP}$  as a whole consists of infinitely many layers of problems, each more difficult than the one which precedes it. It is therefore a remarkable fact that  $\mathcal{NP}$  contains problems which are *maximally difficult*, in the sense that every other  $\mathcal{NP}$ -problem can be polytime reduced to them. We call such problems **complete for  $\mathcal{NP}$  with respect to polytime reductions**, or  $\mathcal{NP}$ -**complete** for short.

By definition, a feasible algorithm for just one  $\mathcal{NP}$ -complete problem would entail that *all* of  $\mathcal{NP}$  can be done in polynomial deterministic time, that is, that  $\mathcal{P} = \mathcal{NP}$ .

**9. Cook’s Theorem.** In 1971, Stephen A. Cook introduced the concept of  $\mathcal{NP}$ -completeness, and proved that SAT is an  $\mathcal{NP}$ -complete problem.<sup>5</sup>

The proof is a little long and I have put it in an appendix. I do not expect you to know its details for the exam. All the same, it is not that hard: the basic idea is to reduce solvability by NTMs to the SAT problem. Here is an outline of it.

Let  $\mathcal{X}$  be an arbitrary problem in  $\mathcal{NP}$  and let  $\mathbf{T}$  be an NTM that solves it in the way described above. In other words, the question  $\mathcal{X}$  for a given input is equivalent to the question of whether  $\mathbf{T}$  has a valid execution starting with that input and finishing with answer Yes. Let  $f \in O(n^k)$  be the polynomial in the length  $n$  of  $s$  that bounds the runtime of  $\mathbf{T}$ .

Given an input string  $s$  for the problem  $\mathcal{X}$ , let  $K$  be the value  $f(n)$  where  $n$  is the length of  $s$ . Imagine a complete list of all the configurations as  $\mathbf{T}$  executes (in any of its non-deterministic possibilities). This will be a big table that has at most  $K$  rows, and also has a number of columns that can be fitted within a number known from  $K$  – because in  $K$  moves the head cannot possibly move more than  $K$  cells to the left or write of the input string. Each entry in this big table will be a tape symbol or a state number.

Instead of saying directly what a given entry is, we can translate into logic by having a set of propositional symbols for that place in the table, one for each tape symbol or state number. For example, if the tape alphabet is  $\{a, b, \sqcup\}$  and there are four states  $\{0, 1, 2, 3\}$ , then for each place in the table we need seven propositional symbols. If the entry there is  $b$ , then its proposition for  $b$  should be true and the rest should be false.

Now the strategy is to write down a huge formula that says that the table entries describe a valid execution of the Turing machine that starts with the given input and ends with the answer Yes. Thus the question of whether there is such a valid execution is equivalent to the satisfiability question for the formula. This is just the reduction we need; and it is polytime.

**10. Example: Reducing a problem to SAT.** Suppose the nodes of a graph<sup>6</sup> are to be coloured by one of the three colours red, blue, or yellow, in such a way that nodes connected by an edge receive different colours. The question of whether such a colouring exists for a given graph is called the “3-colourability problem”. It is clearly a member of  $\mathcal{NP}$ .

I’ll show how to reduce the 3-colourability problem directly to SAT. Cook’s Theorem tells us this is possible, but not how to do it except by constructing a non-deterministic Turing machine and then reducing that to a SAT problem. Such an indirect reduction is usually not practical.

We use the atomic propositions

$$\begin{aligned} \phi_{i,r} &\equiv \text{“node } i \text{ is red”} \\ \phi_{i,b} &\equiv \text{“node } i \text{ is blue”} \\ \phi_{i,y} &\equiv \text{“node } i \text{ is yellow”} \end{aligned}$$

First, we have a formula to express the property that node  $i$  has exactly one of the three colours:

$$(\phi_{i,r} \wedge \neg\phi_{i,b} \wedge \neg\phi_{i,y}) \vee (\phi_{i,b} \wedge \neg\phi_{i,y} \wedge \neg\phi_{i,r}) \vee (\phi_{i,y} \wedge \neg\phi_{i,r} \wedge \neg\phi_{i,b}).$$

<sup>5</sup>At about the same time, Leonid Levin in the Soviet Union independently came up with a concept equivalent to  $\mathcal{NP}$ -completeness, and proved a theorem similar to Cook’s.

<sup>6</sup>This is graph in the sense of graph theory. A graph has a set of nodes, and a set of edges, each joining two nodes.

Writing this out takes polynomial time and space. It has 30 symbols, but we have to count extra for the subscripts such as  $i$  (written in binary) and  $r$ . This is no worse than linear in the number of nodes.

Next, here is a formula to express the property that the colour of node  $i$  is different from the colour of node  $j$ :

$$(\phi_{i,r} \wedge (\phi_{j,b} \vee \phi_{j,y})) \vee (\phi_{i,b} \wedge (\phi_{j,y} \vee \phi_{j,r})) \vee (\phi_{i,y} \wedge (\phi_{j,r} \vee \phi_{j,b})).$$

Again, writing this out is no worse than linear time and space.

Now we make a big formula  $\Phi$  by conjoining (“anding together”) the formulas of the first kind for all values of  $i$ , and the formulas of the second kind for all pairs  $(i, j)$  of nodes joined by an edge. Writing this is no worse than quadratic in the number of nodes and edges. Allowing for the time taken to analyse the input description of the graph, it is still polytime.

To satisfy  $\Phi$  is to find logical values for the proposition symbols so that (i) they determine a colour at each node, and (ii) the colours at the two ends of any edge are different.

Thus we have polytime reduced 3-colourability to SAT. As it happens, the 3-colourability problem is  $\mathcal{NP}$ -complete, so SAT can also be reduced to 3-colourability.

**11. Different notions of reduction.** Most textbooks will require reductions between  $\mathcal{NP}$  problems to be effected by deterministic Turing machines which use only logarithmic additional space. Such machines can be shown to run in polynomial time but it is not known whether the converse also holds.

In general, the more restrictive we are about reductions, the finer are the distinctions we can make between problems, and the smaller is the set of complete problems. However, since we don’t know whether logarithmic space is truly more restrictive than polynomial time, the difference is at this stage purely academic.

**12. More complexity classes.** The classes  $\mathcal{P}$  and  $\mathcal{NP}$  are the most important complexity classes from a practical point of view. Literally hundreds of other classes have been studied as well. In this vast zoo of possibilities the following hierarchy is particularly interesting because it exposes most clearly our ignorance regarding the difference between feasible and intractable problems:

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \text{PSPACE}$$

The class LOGSPACE consists of all those problems which can be solved in logarithmic additional space; NLOGSPACE is the nondeterministic variant of this. Similarly, PSPACE refers to a polynomial amount of extra space. Here the nondeterministic variant is equal to the deterministic one.

For this hierarchy it is known that PSPACE is strictly larger than LOGSPACE, but we know nothing about any of the individual comparisons LOGSPACE  $\subseteq$  NLOGSPACE, NLOGSPACE  $\subseteq$   $\mathcal{P}$ ,  $\mathcal{P} \subseteq$   $\mathcal{NP}$ , or  $\mathcal{NP} \subseteq$  PSPACE. The commonly accepted conjecture is that all inclusions are strict.

**13. Simpler  $\mathcal{NP}$ -complete problems.** When trying to show that a particular problem  $\mathcal{X}$  is  $\mathcal{NP}$ -complete, one does not need to go through another proof such as that for Cook’s Theorem. All we need to do is to show that SAT reduces to  $\mathcal{X}$ . This is because two polytime reductions can be combined, one after the other, to make another polytime reduction. We know that any  $\mathcal{NP}$  problem  $\mathcal{Y}$  polytime reduces to SAT, so combining this with the reduction from SAT to  $\mathcal{X}$  gives us a polytime reduction from  $\mathcal{Y}$  to  $\mathcal{X}$ .

SAT is a problem with quite a nice simple structure. But we can do even better. First of all, we can restrict SAT to formulas in conjunctive normal form – that is to say, they are conjunctions ( $\wedge$ ) of disjunctions ( $\vee$ ) of literals (variables and negated variables). This is because the formula in the proof of Cook’s Theorem needs to be rewritten only slightly to change it into conjunctive normal form. (Note that simply rewriting a formula into conjunctive normal form does not work because the size of the formula may explode exponentially.)

One can then restrict further and stipulate that each disjunction contain at most 3 literals. This is achieved by introducing auxiliary propositional variables. The resulting decision problem is known under the name 3-SAT. Because of the simplicity of its instances it is a good candidate for showing  $\mathcal{NP}$ -completeness via reducibility.

**14. Some  $\mathcal{NP}$ -complete problems**

Graph Theory: Decide whether a given graph contains a closed path (a “cycle”) which visits every node exactly once. (The “Hamiltonian Path Problem”)

Discrete Optimisation: Decide whether a given weighted graph contains a cycle which visits every node at least once, and along which the weights add up to less than a given constant  $K$ . (The “Travelling Salesman Problem”)

Scheduling: Decide whether a set of tasks (with associated running times) can be scheduled on two processors in such a way that all tasks are completed before a given deadline. (The “Multiprocessor Scheduling Problem”)

Robotics: Find the path between two points in three-dimensional space which avoids a given collection of polyhedral obstacles and which is shorter than a given constant  $K$ .

Minesweeper: Richard Kaye (in the School of Maths here at Birmingham) has shown that the “Minesweeper Consistency Problem” is  $\mathcal{NP}$ -complete. This is the decision problem for whether a given configuration of blanks and numbers as in the Minesweeper game can actually arise from a distribution of mines. Kaye cleverly reduced SAT to the Minesweeper problem by representing logic circuits (with wires, AND gates, NOT gates etc.) as Minesweeper configurations. For more details see <http://for.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm>.

More  $\mathcal{NP}$ -complete problems are described in the book Garey & Johnson: *Computers and Intractability: A Guide to the Theory of  $\mathcal{NP}$ -Completeness* (Freeman, San Francisco, 1979).

The problem of finding the factors of a natural number, although clearly in  $\mathcal{NP}$ , does not seem to be  $\mathcal{NP}$ -complete. As yet there is no proof for this, however.

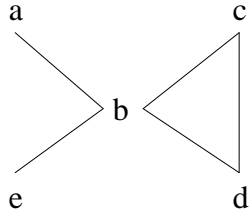
## Exercise Sheet 9

**Quickies** (I suggest that you try to do these *before* the Exercise Class. )

1. For each of the following statements, what would a suitable “witness” be in the sense of checking that the statement is true?
  - (a) “The number  $n$  is composite”
  - (b) “The propositional formula  $\phi$  is satisfiable”
  - (c) “The graph  $G$  is a tree”

**Classroom exercises** (Hand in to your tutor at the end of the exercise class.)

2. Consider this graph:



Let us take a system of 25 propositional variables  $p_{ix}$  ( $1 \leq i \leq 5, x \in \{a, b, c, d, e\}$ ). The intention is to use these to describe a Hamiltonian path so that  $p_{ix}$  means “node  $x$  is the  $i$ th node visited on the path”. Write down propositional formulas to express the following.

- (a) There is at least one  $x$  for which  $p_{1x}$  is true.
- (b) There is at most one  $x$  for which  $p_{1x}$  is true.
- (c) There is at least one  $i$  for which  $p_{ia}$  is true.
- (d) For each  $x$ , if  $p_{2x}$  is true then so is  $p_{3y}$  for some  $y$  joined to  $x$  by an edge in the graph.

By taking together similar formulas, we can make a big formula that says the following.

- (a) For each  $i$  there is exactly one  $x$  for which  $p_{ix}$  is true: hence we obtain a sequence  $x_i$  of nodes in which  $x_i$  is the unique  $x$  for which  $p_{ix}$  is true.
- (b) Each node appears as  $x_i$  for some  $i$ .
- (c) Each  $x_i$  is connected to  $x_{i+1}$  by an edge in the graph.

Hence if the formula is true it tells us a Hamiltonian path through the graph. The same idea works for any graph, so we have reduced the Hamiltonian path problem to SAT. It was obvious that this is possible, since there is obviously a polynomial time checker for the Hamiltonian path problem, so it is in  $\mathcal{NP}$ . It is also true that the problem is  $\mathcal{NP}$ complete, i.e. that SAT can be reduced to it, but that is much harder – see Hopcroft and Ullman. 3

**Homework exercises** (Submit via the correct pigeon hole before next Monday, 2pm.)

3. Consider the the solving problem for factorization: given a natural number  $x$ , find a factor  $y$ . Why is this not a decision problem?

Now consider the following decision problem: for input numbers  $x, m$  and  $z$  (all in binary), it asks whether  $x$  has a factor  $y$ , other than 1 and  $x$ , whose rightmost  $m$  bits agree with  $z$ . Why is this in  $\mathcal{NP}$ ? Why is the special case with  $m = z = 0$  already known to be polynomial?

Show how to polytime reduce the solving problem to this decision problem.

Explain why this shows that if  $\mathcal{P} = \mathcal{NP}$  then factorization would be polynomial. 3

4. You probably know the Towers of Hanoi problem, with  $n$  discs of varying sizes, placed in a pile. (If not, it doesn't matter.) For the well known recursive solution, if  $f(n)$  is the number of steps needed for  $n$  discs then  $f(1) = 1$  and

$$f(n + 1) = 2f(n) + 1$$

Write down the exponential formula for  $f(n)$ . (If you write down the first few values you should be able to guess it; an extra mark if you can prove it from the above equation.)

Suppose a computer executing the algorithm can carry out  $10^9$  steps per second. Roughly how will it take to complete the execution for  $n = 64$ ?

You don't need a calculator. Estimate the answer by using the rule of thumb that  $2^{10} \approx 1000$ . Also, there are about 30 million seconds in a year. 3

**Stretchers** (Problems in this section go beyond what we expect of you in the May exam. Please submit your solution through the special pigeon hole dedicated for these exercises. The deadline is the same as for the other homework.)

*No stretchers this week.*