P, NP and NPC

Course notes for Combinatorial Search

Spring 2007

Peter Bro Miltersen

April 3, 2007

Version 3.1

1 Decision problems and languages

In this note we shall develop the theory of **NP**-completeness. The theory will enable us to show that concrete search and optimization problems are the "least likely" among all "simple" search and optimization problems to have polynomial time algorithms (a notion which will also be made rigorous in this note).

Our first task is to make a restriction of the problems we are going to look at: We shall focus on *decision problems*, i.e., problems for which the output to be computed on a given input (i.e., problem instance) should be either "yes" or "no". Furthermore, we are going to look only on problems where each instance is given by a string over $\{0, 1\}$. Such a problem can be described as a *language*, i.e., a subset L of $\{0, 1\}^*$ with the members of L being the inputs for which the answer to be computed is "yes" and the non-members being the inputs for which the answer to be computed is "no".

The following comments and clarifications should convince us that this seemingly restrictive framework is in fact quite general.

First let us discuss the restriction of inputs to being strings over $\{0, 1\}$. The fact that the alphabet is $\{0, 1\}$ and not some other alphabet is inconsequential: If we have a problem with inputs that we prefer to describe over a different (and most likely bigger) alphabet Σ, we may encode each symbol $\sigma \in \Sigma$ as a string over $\{0, 1\}$ of length $\lceil \log_2 |\Sigma| \rceil$ and then represent a string over Σ by the concatenation of the encodings of its symbols. Indeed, this is the way bigger alphabets such as ASCII and UNICODE are actually represented on real computers. Thus, we are really merely restricting our inputs to be strings over *some* finite alphabet. On the other hand, we are ruling out problems where the input is, say, a vector of arbitrary real numbers, and thus it seems that we are ruling out some of the problems we already looked at, such as network flow problems and linear programming. But, we should remember that strings over $\{0, 1\}$ are in reality the only objects that are actually present in a digital computer, its memory image being itself such a string! So even if we did not make the restriction in our theory, we would have to make it in practice anyway. For instance, even though it may be a useful abstraction to view, say, the simplex algorithm as operating on arbitrary real numbers, we have to make a restriction on the actual set of numbers used when we implement it. A "dirty" but useful such restricted set of numbers would be the set of floating point numbers of some precision, while a cleaner choice would be the set of rational numbers (which will be our choice when we look at problems such a linear program-

ming in this note). In either case, we may represent the restricted set of numbers as strings over a finite alphabet. Also, combinatorial objects such as graphs (used, for instance, to specify a flow networks) are easily representable as strings - and must be, as we *can* give them as inputs to real software. So, demanding a representation of problem instances as strings is not a real limitation. *Which* representation to use *usually* does not matter much - but sometimes it does. This point will be discussed more in the next section, where we define the complexity class $\mathbf{P}$. One standard concrete representation which we will introduce now for convenience is the *pairing function* $\langle \cdot, \cdot \rangle$: If $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$ are strings over $\{0,1\}$, we let $\langle x, y \rangle$ denote the string $x_1 0 x_2 0 \ldots x_{n-1} 0 x_n 011 y_1 0 y_2 0 \ldots y_{m-1} 0 y_m 0$. Note that $\langle x, y \rangle$ is also a string over $\{0,1\}$ and that x and y may be reconstructed from $\langle x, y \rangle$. We generalize the notation to tuples and lists, e.g., if $x = x_1 x_2 \ldots x_n$, $y = y_1 y_2 \ldots y_m$ and $z = z_1 z_2 \ldots z_l$ we let $\langle x, y, z \rangle$ denote the string $x_1 0 x_2 0 \ldots x_{n-1} 0 x_n 011 y_1 0 y_2 0 \ldots y_{m-1} 0 y_m 011 z_1 0 z_2 0 \ldots z_{l-1} 0 z_l 0$ and similar for 4-tuples, 5-tuples, etc. We shall reserve the $\langle \cdot, \cdot \rangle$ notation for pairing functions in this note and not use it to denote inner products. Instead, we write the inner product of vectors c and x as $c^{\mathrm{T}} x$. Also, for a natural number n, we let $b(n) \in \{0,1\}^*$ denote its binary notation.

Given a particular choice on how to represent inputs by strings, a given string may represent no legal input. For instance, if we represent directed graphs by square adjacency matrices and represent the adjacency matrices as $\{0,1\}$-strings by concatenating their rows, only inputs of length $m = n^2$ for some integer n actually represent graphs. We may want to think of an input of non-square length m as being neither a "yes"-instance nor a "no"-instance, but rather being a malformed instance. However when formalizing decision problems as languages we do demand every string to be either a "yes"-instance or a "no"-instance, and we shall simply categorize the malformed instances as "no"-instances. Lumping the malformed instances with the "no"-instances shall work just fine in the theory we shall develop.

The restriction to decision problems or languages, i.e., the restriction to outputs being "yes" or "no" is also less serious than it may first appear. For instance, suppose we really want to have arbitrary Boolean strings as outputs (and as for the inputs, Boolean strings are in reality the only objects digital computers may give as outputs) and want to compute a function $f : \{0,1\}^* \to \{0,1\}^*$. We can then represent f by the decision problem L_f with

$$L_f = \{\langle x, b(j), y \rangle | x \in \{0,1\}^*, j \in \mathbf{N}, y \in \{0,1\}, f(x)_j = y\}.$$

If we can compute f efficiently on a computer, we can also solve the decision problem L_f efficiently. Conversely, if we can solve the decision problem L_f efficiently on a computer, we can also compute f only linearly less efficiently, by computing the output bit by bit. We shall look at efficiency from quite a "coarse" point of view (to be formalized in the next section) so this linear slowdown is inconsequential for our theory. Thus, we shall use the language L_f as a "stand in" for the function f.

More serious is the fact that we are sometimes interested in solving problems that cannot obviously be expressed as computing a *function*, where functions are characterized by having their output uniquely determined from their input. Important examples are the optimization problems we have discussed so far in this course. For instance, take a linear program in standard form: Find $x \in \mathbf{R}^n$ maximizing $c^{\mathrm{T}}x$ that $Ax \leq b, x \geq 0$. There may be *several* different optimal solutions x^* and we don't want to *a priori* rule out any of them, so how can we make a decision problem that may stand in for the problem of solving linear programs in standard form? We shall deal with this in way which is in some sense not quite satisfactory but shall work in this course. Suppose that we are given an optimization problem OPT of the following form.

OPT: "Given an input string defining a set of feasible solutions F and an objective function f, find $x \in F$ maximizing $f(x)$".

We shall associate to OPT the following decision problem, L_{OPT}.

L_{OPT}: "Given an input string defining F and f and a *target value* $v \in \mathbf{Q}$, decide if there is a solution $x \in F$ so that $f(x) \geq v$."

We shall use L_{OPT} as a stand-in for OPT. This is not quite as convincing a stand-in as L_f is for f as we *could* have that L_{OPT} has an efficient algorithm without OPT having an efficient algorithm. However, the goal of our theory is to be able to argue that certain optimization problems OPT do *not* have an efficient algorithm, and this will certainly follow from L_{OPT} having no efficient algorithm, so L_{OPT} will serve its purpose as a stand-in anyway.

As an example of the general ideas above, let us see how we might represent the Traveling Salesman Problem (TSP) by a language L_{TSP}. Recall that a TSP instance is given by a real matrix $(d_{i,j})$ of non-negative distances. As we cannot represent arbitrary real numbers, we restrict our attention to rational numbers $d_{i,j} = p_{i,j}/q_{i,j}$ for integer $p_{i,j}$, $q_{i,j}$. Then, we can define L_{TSP}. As we have a minimization problem rather than a maximization problem, we ask for the existence of a solution of length *smaller* than a given target r/s, rather than larger:

$$
\begin{aligned}
L_{\text{TSP}} \quad = \quad \{\langle \quad & b(p_{0,0}), b(q_{0,0}), b(p_{0,1}), b(q_{0,1}), \ldots, b(p_{0,n-1}), b(q_{0,n-1}), \\
& b(p_{1,0}), b(q_{1,0}), b(p_{1,1}), b(q_{1,1}), \ldots, b(p_{1,n-1}), b(q_{1,n-1}), \\
& \ldots \\
& b(p_{n-1,0}), b(q_{n-1,0}), b(p_{n-1,1}), b(q_{n-1,1}) \ldots, b(p_{n-1,n-1}), b(q_{n-1,n-1}), \\
& b(r), b(s) \ \rangle \\
& |\exists \text{ permutation } \pi \text{ on } \{0, 1, \ldots, n-1\} : \textstyle\sum_{i=0}^{n-1} \frac{p_{\pi(i),\pi((i+1) \bmod)}}{q_{\pi(i),\pi((i+1) \bmod n)}} \leq \frac{r}{s} \ \}
\end{aligned}
$$

2 Turing machines and P

The Turing machine model is a primitive, yet general model of computation. We shall use it in in this section to make precise and completely rigorous the notion of a *polynomial time algorithm.*

A Turing machine consists of

1. A (potentially infinite) bi-directional *tape* divided into *cells* each of which is inscribed with a symbol from some finite *tape alphabet*, Σ. We shall assume that the alphabet includes at least the symbols 0,1 and #, the latter being interpreted as blank. The *position* of a given cell is an integer indicating where on the tape it is placed, i.e., the cell at position -17 is to the immediate left of the cell at position -16.

2. A read/write *head* that at any given point in time t is positioned at a particular cell.

3. A *finite control*, defined by a map

$$\delta : (Q - \{\texttt{accept}, \texttt{reject}\}) \times \Sigma \to Q \times (\Sigma \cup \{\texttt{left}, \texttt{right}\})$$

   where `left` and `right` are two symbols not in Σ and Q is a finite set of *states.* There are three distinguished states of Q, the *start* state `start`, the accepting state `accept` and the rejecting state `reject`.

The finite control defines the operational semantics of the machine in the following way. At any point in time $t \in \mathbf{Z}$, the finite control is in exactly one of the states $q \in Q$, the tape cells are inscribed with a particular sequence of symbols and the head is positioned on a particular cell, inscribed by some symbol $\sigma \in \Sigma$. Put together, we shall refer to these three items as a *configuration* of the machine. If q is either `accept` or `reject`, we say that the configuration is *terminal*, and the configuration of the machine at time

$t+1$ is then defined to be the same as the configuration at time t. If q is `accept` we further call the configuration *accepting* and if q is `reject` we call it *rejecting*. Otherwise, at time $t+1$ we obtain a new configuration in the following way. Suppose $\delta(q, \sigma) = (q', \pi)$ Then, the state of the machine at time $t+1$ is q' and

1. if $\pi \in \Sigma$, the symbol σ in the cell is replaced with the symbol π,
2. if $\pi =$ `left`, the head moves one cell to the left, while the contents of the tape is unchanged,
3. if $\pi =$ `right`, the head moves one cell to the right, while the contents of the tape is unchanged.

We give the Turing machine an input $x \in \{0,1\}^n$ by initially (at time $t = 0$) inscribing the symbol x_i to the cell at position i, for $i = 1, 2, .., n$. All other cells on the tape are left blank (i.e., inscribed #). At time $t = 0$, we position the tape head at the cell at position 0 (i.e., to the immediate left of the cell containing the first symbol of the input). The state of the finite control at time $t = 0$ is `start`.

We say that a Turing machine *accepts* an input x if it, when given the input in the way just described and started eventually reaches an accepting configuration. Similarly we say the the machine *rejects* an input if it eventually reaches a rejecting configuration. We say that a Turing machine *decides* a language $L \subseteq \{0,1\}^*$ (or solves the associated decision problem) if it terminates on all inputs $x \in \{0,1\}^*$ and accepts exactly those in L.

It may seem that the basic nature of the Turing Machine does not make it an adequate model of general computation. However, the generally accepted philosophy is that it *is* adequate. This is captured in the following *thesis*.

Church-Turing thesis Any decision problem that can be solved by some mechanical procedure, can be solved by a Turing machine.

The thesis cannot be "proved" as "some mechanical procedure" is not a rigorous concept. However, much philosophical evidence can be given to support the thesis (but won't be in this course). Also, if one goes through the pain of defining models closer to actual digital computers, one can formally prove that these models decide the same class of languages as Turing machines. Such proofs tend to be rather tedious, but provide further evidence for the reasonable nature of the Church-Turing thesis.

In this course, we are interested in studying *efficient* computation and in particular the distinction between exponential time and polynomial time.

Given a Turing Machine that decides some language, we say that it decides the language *in polynomial time* if there is a fixed polynomial p, so that the number of steps taken on any input x is at most $p(|x|)$ where $|x|$ is the length of x. Finally, we define the *complexity class* $\mathbf{P}$ as the class of languages that are decided in polynomial time by some Turing machine.

Again, it may seem that Turing machines are just too slow to be the basis for a definition intending to capture efficient computability. But also again, the generally accepted philosophy is that they are not, as is captured in the following thesis.

Polynomial Church-Turing thesis A decision problem can be solved in polynomial time by using a reasonable sequential model of computation if and only if it can be solved in polynomial time by a Turing machine.

The Polynomial Church-Turing thesis is what makes the class $\mathbf{P}$ an interesting and robust class: We define it using the Turing machine model to make sure we have a rigorous definition, but it would not be a very interesting definition if the particularities of the somewhat arbitrary Turing machine models were important.

The reasoning behind the Polynomial Church-Turing thesis is the same as the reasoning behind the Church-Turing thesis: Once one gets a feeling for how to program a Turing machine to perform arbitrary computational tasks, one also realizes that while performing a given computation on a Turing machine will require more steps than on more realistic models, the number of steps of the Turing machine will still be bounded by a polynomial in the number of steps taken by the realistic computer. For instance, a Turing machine may simulate the storage of a computer with random access memory by putting the contents of every register of the random access machine on its tape. To perform a single operation of the random access machine (for instance, reading a given register), it will need to *scan* the entire written contents of its tape and this may seem prohibitively slow, but as the part of the tape containing written data (other than the original input) cannot be a segment containing more cells than the number of steps of computation already performed, it will not lead to a superpolynomial slowdown.

Because of the Polynomial Church-Turing thesis, we shall almost never actually construct a Turing machine when we want to argue that some decision problem is in $\mathbf{P}$. Rather, we shall just describe an efficient algorithm for the language using pseudo-code and then refer to the Polynomial Church-Turing thesis.

In addition to the notion of $\mathbf{P}$, which is a class of decision problem, we shall

also need the notion of a polynomial time computable *map* $f : \{0,1\}^* \to \{0,1\}^*$. We say that f is polynomial time computable if the following two properties are true.

1. There is a polynomial p, so that $\forall x : |f(x)| \leq |p(|x|)|$.
2. $L_f \in \mathbf{P}$, where L_f is the decision problem associated with f that was defined in Section 1.

By the Polynomial Church-Turing thesis, whether a particular decision problem is decidable in polynomial time or not does not depend on the particularities of the Turing machine model, but we may fear that it depends on the way we formalize problems as languages, i.e., on the way that instances are represented as strings.

In particular, consider the following situation. Given a set S of objects intended as the set of instances for some decision problem $f : S \to \{\texttt{yes}, \texttt{no}\}$ (for instance, S could be the set of all finite directed graphs) and two different ways of representing S as binary strings, i.e., two different injective maps $\pi_1 : S \to \{0,1\}^*$ and $\pi_2 : S \to \{0,1\}^*$ (for instance, π_1 could be an adjacency matrix representation and π_2 could be an edge list representation). We could then represent f as a language by either $L_1 = \{x | f(\pi_1^{-1}(x)) = \texttt{yes}\}$ or $L_2 = \{x | f(\pi_2^{-1}(x)) = \texttt{yes}\}$ and it is not clear, *a priori*, that $L_1 \in \mathbf{P} \Leftrightarrow L_2 \in \mathbf{P}$. To eliminate this fear, we introduce the following notions: We say that a representation π is *good* if $\pi(S)$ is in $\mathbf{P}$, i.e., if it can be decided efficiently if a given string is a valid representation of an object. We say that the representations π_1 and π_2 are *polynomially equivalent* if there are polynomial time computable maps r_1 and r_2 translating between the representations, i.e., for all $x \in S$, $\pi_1(x) = r_1(\pi_2(x))$ and $\pi_2(x) = r_2(\pi_1(x))$.

Proposition 1 *If π_1 and π_2 are good representations that are polynomially equivalent, then $L_1 \in \mathbf{P} \Leftrightarrow L_2 \in \mathbf{P}$.*

We leave the proof of Proposition 1 as Exercise 4. Also, the reader may easily establish that several alternative ways of representing standard objects such as graphs, sets and numbers are in fact good and polynomially equivalent. This is established in Exercise 5.

An important exception is this. We may choose to represent numbers in *unary* notation, i.e., to represent the number n as the string containing n copies of the symbol 1. Let this string be called $u(n)$. Unary representation u is *not* polynomially equivalent to the usual binary representation b (Exercise

5). The binary representation b is the default one when we discuss whether a particular decision problem is in $\mathbf{P}$, but by in addition looking at the problem when numbers are represented in unary, we may gain additional information about its computational properties. In particular we are going to use the following terminology. Let a problem involving integers be given and let L be a language representing the problem using unary notation to represent the integers. If L is in $\mathbf{P}$, then the problem is said to have a *pseudopolynomial time algorithm*. Note that since the unary representation of a number is longer than the binary and since we measure the complexity of an algorithm as a function of the length of the instance, it is easier for a problem to have a pseudopolynomial time algorithm than a polynomial time algorithm.

3 NP and the P vs. NP problem

This course is about problems that could be solved by an exhaustive search through an implicitly represented set of possible solutions - only doing this would be prohibitively slow. The class $\mathbf{NP}$ is intended to formalize this notion (for the case of decision problems).

$\mathbf{NP}$ is defined to be the class of languages L for which there exists a language $L' \in \mathbf{P}$ and a polynomial p, so that

$$\forall x: \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq p(|x|) \wedge \langle x, y\rangle \in L']$$

To understand the definition, one should think of the set of binary strings of length at most $p(|x|)$ as representing potential solutions to the search problem associated with instance x. With this interpretation, $\langle x, y\rangle \in L'$ means that the potential solution y is indeed a valid solution to the problem. We can think of the search problems we may capture in this way as "simple" search problems in the following sense:

1. We require that $L' \in \mathbf{P}$, i.e., we want it to be possible to efficiently check that a given solution y indeed is a valid solution for the instance x.

2. We require that the potential solutions have length at most $p(|x|)$, i.e., length polynomial in the problem instance.

Thus, for a language L in $\mathbf{NP}$, we *could* decide if $x \in L$ by going though all $2^{p(|x|)+1} - 1$ possible values of y and for each of them checking if $\langle x, y\rangle \in L'$

but we probably don't want to perform this exhaustive search, as this would take exponential time.

As an example, let us see that the language L_{TSP} we defined earlier is in **NP**. We may decide if an instance x containing an $n \times n$ distance matrix is in L_{TSP} by going through all permutations on $\{0, 1, 2, \ldots, n-1\}$ and checking if one of the corresponding traveling salesman tours has total length at most the target value of the instance. Thus, if we define L' as

$$\begin{aligned} L' \;=\; & \{\langle\langle b(p_{0,0}), b(q_{0,0}), b(p_{0,1}), b(q_{0,1}), \ldots, b(p_{0,n-1}), b(q_{0,n-1}), \\ & b(p_{1,0}), b(q_{1,0}), b(p_{1,1}), b(q_{1,1}), \ldots, b(p_{1,n-1}), b(q_{1,n-1}), \\ & \ldots \\ & b(p_{n-1,0}), b(q_{n-1,0}), b(p_{n-1,1}), b(q_{n-1,1}) \ldots, b(p_{n-1,n-1}), b(q_{n-1,n-1}), \\ & b(r), b(s)\rangle,\ \langle \pi(0), \pi(1), \ldots, \pi(n-1)\rangle\rangle \\ & \mid \textstyle\sum_{i=0}^{n-1} \frac{p_{\pi(i),\pi((i+1) \bmod)}}{q_{\pi(i),\pi((i+1) \bmod n)}} \leq \frac{r}{s}\ \} \end{aligned}$$

we get

$$\forall x: \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq |x|^2 \wedge \langle x, y\rangle \in L']$$

so L_{TSP} is in **NP**. Other natural examples of problems in **NP** are given in the exercises and we shall also meet many other problems later. In fact, we clearly have that $\mathbf{P} \subseteq \mathbf{NP}$. Indeed, if $L \in \mathbf{P}$, we can just define L' by $\langle x, y\rangle \in L'$ if and only if $x \in L$, i.e., we can simply ignore the y-part of the input when deciding L. Thus, **NP** definitely contain some very easy search problems. The interesting problem is whether the converse containment hold: Do *all* simple search problems have polynomial time algorithms, i.e., algorithms that avoid doing an exhaustive search through the space of all possible solutions or a space of size close to the size of this space? This question was first[1] asked by Cook in 1972 and is still unanswered.

Open Problem: Is **P**=**NP**?

The problem is on the list of the Clay Mathematics Institute list of Millennium Problems (`http://www.claymath.org/millennium/`) which in addition to indicating its recognized status as an important open problem means that there is a \$1 million dollar prize for solving it!

While we do not know for sure, most people believe that there *are* problems in **NP** that are not in **P**, i.e., simple search problems for which there is no

[1]To be precise, Cook's 1972 publication was the first published paper asking the question. But in fact the question was already asked in the 1950's in a private letter from one famous mathematician, Kurt Gödel (of the incompleteness theorem of logic) to another, John von Neuman (co-inventor of linear programming and zero sum games).

efficient alternative to searching through an exponentially big search space of possible solutions. Their belief comes from the generality of the notion of **NP**. In particular, consider the activity of mathematics, a substantial part of which consists of finding proofs of theorems. While proofs as communicated between mathematicians are informal, it is a well-established principle that a proof should be in principle formalizable. Also, set theory, as formalized by the formal system ZFC (Zermelo-Fränkel with the Axiom of Choice) is generally accepted as a formal system strong enough to capture all of mainstream mathematics. Here, it is not important exactly what this formal system is, as we shall only use the following properties of it in our discussion: Theorems and proofs in ZFC are strings over some finite alphabet Σ. By encoding each symbol of Σ as a string over $\{0, 1\}$, we can also think of theorems and proofs in ZFC as Boolean strings. Furthermore, given an alleged theorem string t and an alleged proof string p, we can decide in polynomial time in $|t| + |p|$ if p really is a proof of t, i.e., formal proofs can be *checked* efficiently. The compelling evidence against **P**=**NP** is then this:

Proposition 2 *If* **P**=**NP***, then there is an algorithmic procedure that takes as input a formal statement t of ZFC and, if this statement has a proof in ZFC of length n, the procedure terminates in time polynomial in $|t| + n$ outputting the shortest proof of t.*

Proof Define MATH $= \{\langle t, u(n), s\rangle |$ There is a ZFC-proof of length at most n proving statement t and the proof begins with the string $s\}$ where u is the unary notation of integer n. We have that MATH $\in$ **NP**, as we could search through all strings s' of length at most $n - |s|$ and check for each of them if $s \cdot s'$ is a valid proof of t. Now, if **P**=**NP**, we also have that MATH is in **P**. This means that there is a polynomial time algorithm deciding MATH.

Now, let λ denote the empty string. Given t, run the efficient algorithm for MATH on inputs $\langle t, u(1), \lambda\rangle, \langle t, u(2), \lambda\rangle, \langle t, u(3), \lambda\rangle, \ldots$ until we find an n so that $\langle t, u(n), \lambda\rangle$ is in MATH. We now know that the shortest proof of t has length n. We can now find a proof of length n by a "binary search" procedure: We first check if $\langle t, u(n), 0\rangle$ is in MATH. If it is, we know that there is a proof of length n starting with 0 and we may then start figuring out the second symbol of the proof by checking if $\langle t, u(n), 00\rangle$ is in MATH. If $\langle t, u(n), 0\rangle$ is not in MATH, we know that there is a proof of length n starting with 1 and we then start to figure out the second symbol of the proof by checking if $\langle t, u(n), 10\rangle$ is in MATH, etc. Each time we run the algorithm for MATH we learn another symbol of the proof. Thus, by running the algorithm for MATH n additional times, we end up knowing the shortest proof of theorem t and can give it as output. This completes the proof of the proposition.

Why is Proposition 2 evidence against **P**=**NP**? The reason is this. Often proofs found by mathematicians are extremely ingenious, containing non-trivial and deep ideas and we celebrate these proofs and the people who find them (in particular, we may give then one million dollar awards, like the Clay awards mentioned above). Even so, relatively short and elegant proofs may go unnoticed for centuries. We certainly do *not* suspect that finding such proofs may be automatized in a way that is *efficient in the worst case.*, i.e., that a Turing machine can find the shortest proof of any theorem in time polynomial time in the length of the proof, no matter how tricky, deep and ingenious it is! Hence we believe that $\mathbf{P} \neq \mathbf{NP}$.

4 Reductions and the complexity class NPC

Our interest in **NP** comes from the fact that the class contains many problems we would like to solve. If we assume that $\mathbf{P} \neq \mathbf{NP}$, some of these problems cannot be solved by an efficient (i.e., polynomial time) algorithm. On the other hand, many problems in **NP** are easy, as **P** *is* a subset of **NP**. If we have a particular problem at hand and have worked on it for a while without coming up with an efficient algorithm for the problem, we may begin to suspect that it has no efficient algorithm. The concept of **NP**-*completeness* to be introduced in this section helps us converting this suspicion into a proven fact (under the assumption that **P**≠**NP**) and hence avoiding wasting time trying to construct a worst case efficient algorithm that doesn't exist!

We first need to formalize the notion of a *reduction.* Given two languages L_1 and L_2, we say that *there is a reduction from* L_1 *to* L_2 or alternatively that L_1 *reduces to* L_2 and write $L_1 \leq L_2$ if there is a polynomial time computable function r (the reduction) so that for all $x \in \{0,1\}^*$, we have that $x \in L_1$ if and only if $r(x) \in L_2$.

The $\leq$-notation suggests a partial order, which is slightly misleading: It is *not* true that $L_1 \leq L_2$ and $L_2 \leq L_1$ implies $L_1 = L_2$ (why not?). On the other hand, reductions *do* satisfy the transitivity property:

Proposition 3 *If* $L_1 \leq L_2$ *and* $L_2 \leq L_3$ *then* $L_1 \leq L_3$.

Proof We have a polynomial time computable map r_1, r_2 so that for all x, we have $x \in L_1$ if and only if $r_1(x) \in L_2$ and for all y, we have $y \in L_2$ if and only if $r_2(y) \in L_3$. Hence, for all x, $x \in L_1$ if and only if $r_2 \circ r_1(x) \in L_3$. Furthermore $r_2 \circ r_1$ is a polynomial time computable map and hence $L_1 \leq L_3$.

A reduction r from L_1 to L_2 is, intuitively, just a translation of L_1 instances into L_2 instances: If we want to know the answer to an L_1-instance x but only have an algorithm for L_2 running in time $q(n)$ on inputs of length n, we may compute $r(x)$ and use the algorithm to get the answer using time $q(|r(x)|)$. Appealing to the Polynomial Church-Turing thesis, we have established the following formal statement:

Proposition 4 *If $L_1 \leq L_2$ and $L_2 \in \mathbf{P}$ then $L_1 \in \mathbf{P}$.*

We may think of L_2 as being a more general language than L_1 and hence less likely to be in **P**. Now suppose a language L_2 has the property that for *all* $L_1 \in \mathbf{NP}$ we have that L_1 reduces to L_2. Intuitively, such a language is less likely to be in **P** than any languages in **NP**. Formally, we call such a language **NP**-*hard* and capture the intuition in the following statement.

Proposition 5 *Let L be an* **NP**-*hard language. If* $\mathbf{P} \neq \mathbf{NP}$ *then* $L \notin \mathbf{P}$.

Proof That L is **NP**-hard means that $\forall L' \in \mathbf{NP} : L' \leq L$. If we assume to the contrary that L is in **P**, then by Proposition 4, all $L' \in \mathbf{NP}$ is in **P** and hence **P**=**NP**.

NP-hard languages do not necessarily lie in **NP**. But we are particularly interested in those that do: We define the class **NPC** of **NP**-*complete* languages to be those languages in **NP** that are **NP**-hard.

Proposition 6 *Let $L \in \mathbf{NPC}$. Then* **P**=**NP** *if and only if L is in* **P**.

Proof Proposition 5 gives the "if" direction. The "only if" direction follows from L being in **NP**.

Getting back to the situation described in the beginning of the section: Having worked on a particular problem in **NP** and failed to find an efficient algorithm, we might instead try to establish that the problem is in **NPC**. If we do, and if we believe that **P**$\neq$ **NP** (perhaps convinced by the evidence in the preceeding section), we are now satisfied that no efficient algorithm exists. Even if we have no opinion about whether **P**$\neq$**NP**, we at least know that constructing an efficient algorithm for our problem is a non-trivial task: Indeed, if we construct an algorithm and prove that it has polynomial worst case time complexity, we have proved that **P**=**NP** and will be awarded one million dollars for our achievement by the Clay institute!

It is therefore of great interest to prove that natural problems in **NP** that we would like to solve but do not know an efficient algorithm for are **NP**-complete. A priori, it is not clear that this would be the case for very many problems (it is not perhaps not even clear that it is the case for *any* problem) and hence it is not clear that the definition of **NPC** would be very useful. Fortunately, it turns out that a substantial fraction of the natural problems in **NP** that we do not know how to solve *are* **NP**-complete, and we shall be able to prove this. However, if we use the definition of **NPC** for such a proof, we must prove that *any* problem in **NP** reduces to the problem we are interested in, and this may seem quite awkward to prove. To prove that a concrete problem is **NP**-hard, we shall usually use the following lemma instead which makes it to sufficient to establish a *single* reduction.

Lemma 7 *If L_1 is* **NP***-hard and $L_1 \leq L_2$ then L_2 is* **NP***-hard.*

Proof Since L_1 is **NP**-hard, every language L in **NP** reduces to L_1. Since L_1 reduces to L_2, we may use Proposition 3 and conclude that every language L in **NP** reduces to L_2. Hence L_2 is **NP**-hard, as desired.

Thus, when we establish that a certain problem is **NP**-complete, we can add it to a "mental database" of such problems, and it becomes easier to establish **NP**-completeness of new problems L in **NP**: We just need to find one problem in the database that we can reduce to L and we may appeal to Lemma 7. Indeed, this is how we are going to operate. The tricky thing is to get the database started: We must establish one natural "mother" problem to be **NP**-complete. This was first done by Cook, in *Cook's Theorem* to be established in Section 6. To state and prove it, however, we must first introduce the notion of a Boolean circuit.

5 Boolean Circuits

For notational convenience, we shall make the well-known identification of the Boolean value `true` with 1 and the Boolean value `false` with 0 in this and following sections.

A *Boolean circuit* with n input gates and m output gates is a directed, acyclic graph $G = (V, E)$. The vertices of V are called *gates*. Each gate has a *label* which indicates the type of the gate. A label is either taken from the set of *function* symbols $\{AND, OR, NOT, COPY\}$, from the set of *constant* symbols $\{0, 1\}$ (representing `false` and `true`), or from the set of *variable*

symbols $\{X_1, X_2, \ldots, X_n\}$. For each j, at most one gate is labeled X_j. The gates labeled with a variable symbol are called input gates. We shall call gates labeled AND for AND-gates and use similar terminology for the other types of function gates. The arcs in E are called *wires*. If there is a wire from gate u to gate v, we say that u is an input to gate v. The following constraints must be satisfied:

1. Each AND-gate and OR-gate has exactly two inputs.
2. Each NOT-gate and COPY-gate has exactly one input.
3. The input gates are sources in G.

Finally, m of the gates are designated as *output gates*; we call these $o_1, o_2, \ldots, o_m$.

A Boolean circuit C defines or *computes* a Boolean function from $\{0,1\}^n$ to $\{0,1\}^m$ in the following way: Given a vector $x \in \{0,1\}^n$, we can assign the value x_i to each gate labeled X_i. The gates labeled 0 (1) are assigned the Boolean values 0 (1). Then, we iteratively assign values to the rest of the gates using the following rules.

1. Whenever the two inputs of an AND-gate have been assigned values, we assign the boolean AND of these two values to the gate.
2. Whenever the two inputs of an OR-gate have been assigned values, we assign the boolean OR of these two values to the gate.
3. Whenever the input of a NOT-gate has been assigned a value, we assign the boolean negation of this value to the gate.
4. Whenever the input of a COPY-gate has been assigned a value, we assign the same value to the gate.

Since the circuit is acyclic and the only sources of the graph are assigned values when we begin, each gate in the circuit will eventually be assigned a value. The value of the function on the input x is then the vector $y \in \{0,1\}^m$ obtained by collecting the values assigned to the output gates $o_1, o_2, \ldots, o_m$. We refer to the entire process as *evaluating* the circuit on input x.

We shall abuse notation slightly and use the name C to denote both the circuit and the function it is computing, i.e., we shall write $C(x) = y$.

Any Boolean function can be computed by some circuit:

Lemma 8 *For any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, there is a circuit C, so that $\forall x \in \{0,1\}^n : C(x) = f(x)$.*

Proof First, we can assume without loss of generality that $m = 1$, as we can combine m 1-output circuits to get an m-output circuit. We also notice that we may view a Boolean *formula* such as $(X_1 \vee X_2) \wedge X_3$ as a special kind of one-output Boolean circuit: One where each function gate is the input to at most one other gate. Indeed, the graph induced by the function gates of such a circuit is a tree, and we may identify it with the parse tree of the formula. Thus, we just have to show that any Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be defined by a Boolean formula with operations $\vee, \wedge\ \neg$ and variables $X_1, X_2, \ldots, X_n$.

Given $f : \{0,1\}^n \to \{0,1\}$. For each $x^* \in f^{-1}(1)$, we can define a formula g_{x^*} which evaluates to 1 on input x^* and to 0 on all inputs, namely the conjunction (i.e., AND) of all variables that are `true` in x^* and the negations of all variables that are `false` in x^*. For instance, if $x^* = (1,1,0)$, we have $g_{x^*} \equiv (X_1 \wedge X_2) \wedge \neg X_3$. The desired formula for f is then the disjunction (i.e., OR) of g_{x^*} over all $x^* \in f^{-1}(1)$. This completes the proof.

Thus, we may regard circuits as low-level, but completely general computational devices. However, unlike a Turing Machine, a circuit takes inputs of a *fixed* input length only. Circuits serve an important role in the theory of NP-completeness because of their dual role of computational devices and as combinatorial objects that may be used as inputs to natural search and optimization problems.

By the *size* of a circuit we shall mean the number of gates in the circuit. The size of a circuit is a measure of the time required to evaluate the circuit on an input. The following important lemma connects the time measure of the Turing machine model to the size measure of the circuit model.

Lemma 9 *Let any Turing machine M running in time at most $p(n) \geq n$ on inputs of length n, where p is a polynomial, be given.*

Then, given any fixed input length n, there is a circuit C_n of size at most $O(p(n)^2)$ so that for all $x \in \{0,1\}^n$, $C_n(x) = 1$ if and only if M accepts x.

Furthermore, the function mapping 1^n to a description of C_n is polynomial time computable.

Before we begin the proof, let us convey its idea by noting that the statement of the lemma is in no way surprising: A Turing machines is a primitive version of a real computer and real computers are actually *made* of Boolean circuits.

Thus, we really should just build a Turing machine using standard electronic components[2]. The main obstacle for such a construction is the fact that the circuits we consider are *acyclic*, while the circuits used to build real computers contain feedback loops. Feedback loops are necessary in real computers for practical reasons: Real computers have to operate without any (or only a very large) a priori upper bound on the computation time and we want to operate them interactively and repeatedly give them new inputs and receive new outputs, etc. For this to be possible the gates of a real computer must be *reused* at each clock cycle, i.e., at each step of the computation. Here, we shall be able to do without feedback loops in the circuit simply by having a separate collection of gates for each step of the computation. As we know there will be at most $p(n)$ steps, this will make the size of our circuit only a polynomial factor larger than if we were allowed to use feedback loop.

With this discussion in mind, we proceed with the proof.

Proof Consider the computation of M on some input of x of length n. As M runs in time $p(n)$ and starts with the tape head in the cell at position 0, it cannot touch any tape cell other than those at position $-p(n), .., 0, .., p(n)$ during the computation. Any other cell will never see the head and will contain the blank symbol # throughout the computation.

For a fixed input x, and each position $i \in \mathbf{Z}$ (though only positions between $-p(n)$ and $p(n)$ will be interesting) and each point in time $t \in \{0, \ldots, p(n)\}$, let us define $c_{t,i}$ to be the following collection of information, which we shall call a *cell state vector*.

1. The symbol written in the tape cell at this point in time.
2. Whether or not the tape head is pointing to the tape cell at this point in time.
3. *If* the tape head is indeed pointing to the tape cell at this point in time, we make a note of the state of the finite control.

A cell state vector is a finite amount of information and we may encode it, in some arbitrary but fixed way, as a bit string $c_{t,i} \in \{0,1\}^s$, where s is some number that depends on M only (and not on x nor even $n = |x|$).

Now the crucial observation is the following:

[2]And since this is the task at hand, the reader will now appreciate that we took the simple Turing machine model as our model of computation, rather than a model closer to realistic computers!

Observation: Suppose we don't know x (or even $n = |x|$). Then, for $t > 1$, we can determine $c_{t,i}$ from $c_{t-1,i-1}$, $c_{t-1,i}$ and $c_{t-1,i+1}$. In other words, there is a fixed Boolean function $h : \{0,1\}^{3s} \to \{0,1\}^s$, depending on M only (and not x nor $n = |x|$), so that $c_{t,i} = h(c_{t-1,i-1} \cdot c_{t-1,i} \cdot c_{t-1,i+1})$

The validity of the observation follows directly from the operational semantics of the Turing machine model. By Proposition 8, there is a circuit D computing h.

We shall also need the following circuits:

- A circuit E computing $E : \{0,1\} \to \{0,1\}^s$ so that $E(b)$ is a cell state vector representing a cell containing the symbol $b \in \{0,1\}$ and not containing the tape head. By Proposition 8, such a circuit E exists.
- A circuit F computing $F : \{0,1\}^s \to \{0,1\}$ so that $F(y) = 1$ if and only if y is a cell state vector representing a cell that does hold the tape head and with the finite control of the Turing machine in the accepting configuration. Again, by Proposition 8, such a circuit F exists.

The circuit C_n we have to construct to prove the lemma will essentially obtained simply by gluing together $(2p(n)+1)p(n)$ copies of D, n copies of E and $(2p(n)+1)$ copies of F and letting the output of C_n be a disjunction of the outputs of the copies of F. The formal details of the construction that now follows will be somewhat tedious (a completely formal proof that the construction is correct would be even more tedious and is omitted). The reader is at this point invited to convince himself that the construction of C_n can be done using the component just defined, by examining Figure 1. The formal definition of C_n follows.

We are going to need some constant gates. In particular, we define a collection B of constant gates B_i, $i = 1, ..s$ so that B_i holds the i'th bit in the cell state vector representing a blank cell without the head.

The circuit C_n will contain $(2p(n)+1)p(n)$ subcircuits $G_{t,i}$, one subcircuit for each position $i \in \{-p(n), \ldots, 0, \ldots, p(n)\}$ and each point in time $t \in \{0, 1, \ldots, p(n)\}$. The subcircuit $G_{t,i}$ is intended to compute the state vector $c_{t,i}$ and is defined in the following way.

1. For $t = 0$ and any i between 1 and n, the subcircuit $G_{t,i}$ is a copy of E but with its single input gate replaced with a COPY gate taking its input from input gate X_i.
2. For $t = 0$ and $i = 0$ we make $G_{0,0}$ a collection of constant gates B',

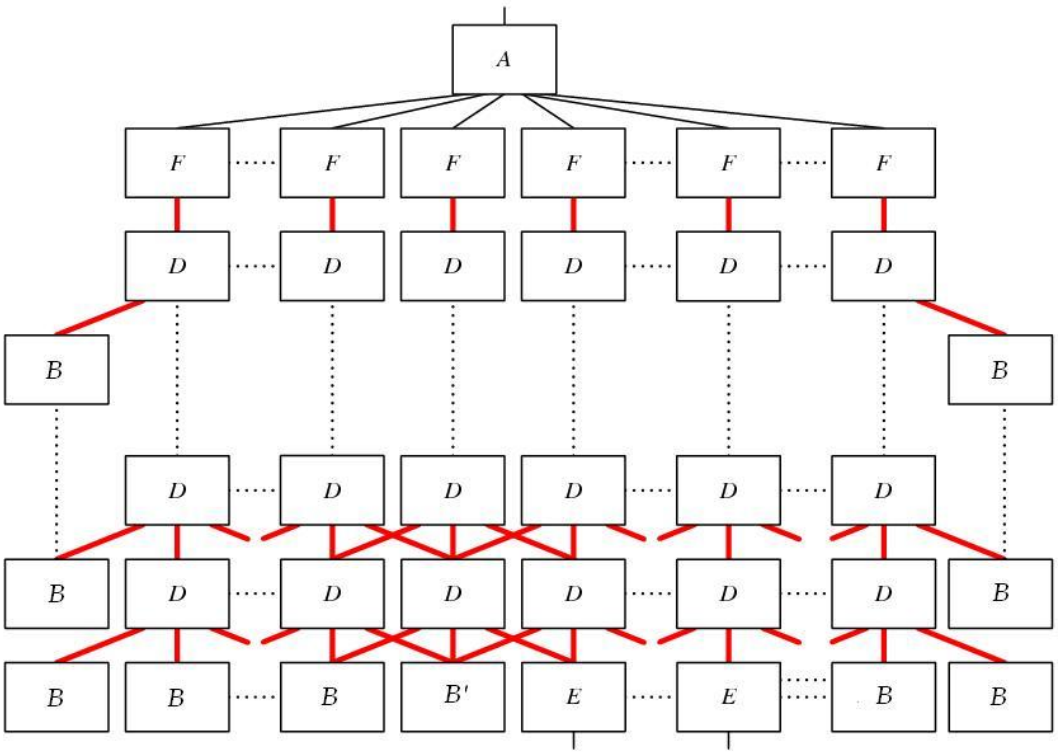

Figure 1: The circuit C_n. Red lines represent "busses" containing s wires. Thanks to Johnni Winther for preparing the figure.

together representing the cell state vector of a blank cell that *does* contain the head and with the finite control in the `start` state.

3. For $t = 0$ and $i \notin \{0, 1, \ldots, n\}$, we make $G_{0,i}$ a copy of B, i.e. a collection of constant gates representing the cell state vector of a blank cell that does not contain the tape head.

4. For $t \geq 1$ and any i between $-p(n)$ and $p(n)$, the subcircuit $G_{t,i}$ is a copy of D but with input gates X_i replaced with COPY-gates in the following way:

   - Each COPY-gate replacing input gate X_i, $i \in \{1, \ldots, s\}$ takes as input the i'th output gate of $G_{t-1,i-1}$, *unless* $i - 1 < -p(n)$, in which case we let the gate take as input B_i.
   - Each COPY-gate replacing input gate X_{s+i}, $i \in \{1, \ldots, s\}$ takes as input the i'th output gate of $G_{t-1,i}$,
   - Each COPY-gate replacing input gate X_{2s+i}, $i \in \{1, \ldots, s\}$ takes as input the i'th output gate of $G_{t-1,i+1}$, *unless* $i + 1 > p(n)$, in which case we let the gate take as input B_i.

We now add $2p(n) + 1$ copies of F, named $H_j, j \in \{-p(n), \ldots, p(n)\}$ to the circuit C_n in the following way: We replace input gate X_i of H_j by a COPY-gate, copying the i'th output gate of the subcircuit $G_{p(n),j}$. Finally, we construct a circuit A computing the OR of $2p(n) + 1$ inputs - such a circuit can be made using $2(2p(n) + 1) - 1$ binary OR-gates arranged in a tree - and replace all the input gates of this circuit with COPY-gates, copying the output gates of H_j, $j \in \{-p(n), \ldots, p(n)\}$.

As output gate of C_n, we retain the single output gate of A (i.e., we do not make the output gates of all the other subcircuits into output gates of C_n).

This completes the construction of C_n. Clearly, the size of C_n is as desired, by construction. Also by construction, C_n has the desired property: It outputs 1 on input x if and only if the machine M accepts x. Furthermore, the completely regular nature of C_n makes it easy to make an efficient algorithm that outputs a representation of C_n in time $n^{O(1)}$ given n: Such an algorithm can have descriptions of the circuits D, E and F built in and then merely has to combine the the correct number of copies of these circuits in the correct way which is a simple task. Appealing to the Polynomial Church-Turing thesis, we have thus established that the function mapping 1^n to a description of C_n is polynomial time computable. This completes the proof of the lemma.

6 Cook's Theorem

Given a set of symbols denoting Boolean variables, a *literal* is a variable or its negation. A *clause* is a disjunction of a number of literals. For instance, $(X_1 \vee X_2 \vee \neg X_3)$ is a clause involving the literals X_1, X_2 and $\neg X_3$. A *Conjunctive Normal Form (CNF) formula* is a conjunction of a number of clauses. For instance, $(X_1 \vee X_2 \vee \neg X_3) \wedge (X_3 \vee \neg X_1)$ is a CNF formula. The SATISFIABILITY PROBLEM or SAT is the following decision problem: Given a CNF formula, decide if there is an assignment of truth values to the variables of the formula so that the formula evaluates to true. Such an assignment is referred to as a *satisfying assignment* of the formula. For instance, for the formula $(X_1 \vee X_2 \vee \neg X_3) \wedge (X_3 \vee \neg X_1)$ the answer is "yes", as, for instance, the assignment $X_1 = X_2 = X_3 = \texttt{true}$ makes the formula evaluate to `true`. On the other hand, for the formula $(X_1 \vee \neg X_2) \wedge (X_2 \vee \neg X_1) \wedge (X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_2)$, the answer is "no", as all truth assignments make this formula evaluate to `false`.

Historically, SAT is the mother of all **NP**-complete problem we alluded to in Section 4:

Theorem 10 (Cook, 1972) SAT $\in$ **NPC**.

Cook proved the theorem by directly showing that all problems in **NP** reduces to SAT. Having already studied Boolean circuits, we can make a somewhat simpler proof by first showing that all problems in **NP** reduces to a related problem, CIRCUIT SAT (Theorem 11) and then showing that CIRCUIT SAT reduces to SAT (Proposition 12). Cook's theorem then follows from Lemma 7 and the obvious fact that SAT is in **NP**. Thus, in these notes, it is actually CIRCUIT SAT that becomes the mother of all **NP**-complete problems. CIRCUIT SAT is the following decision problem: Given a Boolean circuit C, is there a vector x so that $C(x) = 1$? That is, is there an assignment of Boolean values to the input gates of the circuit that will make the output gate of the circuit evaluate to 1? Such an assignment shall be referred to as a *satisfying assignment* of the circuit. Since a Boolean formula may be regarded as a special kind of Boolean circuit, as explained in the proof of Lemma 8, SAT may be regarded as a special case of CIRCUIT SAT.

Theorem 11 CIRCUIT SAT $\in$ **NPC**

Proof CIRCUIT SAT is clearly in **NP**. We need to show that it is **NP**-hard. Let L be a language in **NP**. We must show that there is a polynomial

time reduction r so that

$$\forall x : x \in L \Leftrightarrow r(x) \in \text{CIRCUIT SAT},$$

Since L is in **NP**, by definition there is a language L' in **P** and a polynomial p, so

$$\forall x : \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq p(|x|) \wedge \langle x, y\rangle \in L']$$

Now we may define the reduction r as follows. The reduction should map instances of L to instances of CIRCUIT SAT, i.e., to descriptions of circuits. Given input x, the value of $r(x)$ is going to be a description of a circuit C which may be written as a disjunction $C \equiv D_0 \vee D_1 \vee D_2 \ldots D_{p(|x|)}$, i.e., the reduction should construct $p(|x|)$ subcircuits and combine them using $p(|x|) - 1$ OR-gates. Each subcircuit D_i should take i Boolean inputs and should evaluate to 1 on input $y \in \{0,1\}^i$ if and only if $\langle x, y\rangle \in L'$. If we can achieve this, we clearly have $x \in L \Leftrightarrow C \in$ CIRCUIT SAT, as desired.

The subcircuit D_i is defined as follows. Let M be a Turing machine deciding L' in polynomial time. From Lemma 9, we have an efficient algorithm that given an input length n, outputs a circuit C_n so that for all z, y with $|\langle z, y\rangle| = n$, we have that $C_n(\langle z, y\rangle) = 1$ if and only if M accepts $\langle z, y\rangle$. Let $n = |\langle x, 1^i\rangle| = 2(|x| + i) + 2$. The circuit C_n takes an input $\langle z, y\rangle$. By our definition of $\langle \cdot, \cdot \rangle$, the circuit C_n reads the input z using the input gates labeled $X_1, X_3, X_5, \ldots, X_{2|z|-1}$ Now we replace these input gates with constant gates so that input gate X_{2i-1} is replaced with a constant gate representing the bit x_i. Furthermore, we replace input gates labeled $X_2, X_4, \ldots,$ with constant gates labeled 0 and input gates labeled $X_{2|x|+2}$ and $X_{2|x|+3}$ with constant gates labeled 1. Informally speaking, we hardwire the input x into the circuit. The circuit we obtain in this way is the circuit D_i and it clearly has the right property.

The reduction r should construct the subcircuits $D_0, D_1, \ldots, D_{p(|x|)}$, combine them using OR-gates and output a representation of the resulting circuit C. Appealing to the Polynomial Church-Turing thesis, we conclude that r is a polynomial time computable map and the proof of the theorem is complete.

Proposition 12 CIRCUIT SAT $\leq$ SAT

Proof Given a one-output circuit C we should define a CNF formula $f = r(C)$ so that f has a satisfying assignment if and only if C does and so that r is a polynomial time computable map.

The CNF formula f has a variable for each gate g of C. For convenience, we shall abuse notation slightly and also refer to the variable of f as g.

The clauses of f are defined as follows:

For each AND-gate g of C taking as input gates h_1 and h_2 we add a set of clauses to f logically equivalent to the statement $g \Leftrightarrow (h_1 \wedge h_2)$, namely the clauses:

$$(\neg g \vee h_1) \wedge (\neg g \vee h_2) \wedge (g \vee \neg h_1 \vee \neg h_2).$$

For each OR-gate g of C taking as input gates h_1 and h_2 we add a set of clauses to f logically equivalent to the statement $g \Leftrightarrow (h_1 \vee h_2)$, namely the clauses:

$$(g \vee \neg h_1) \wedge (g \vee \neg h_2) \wedge (\neg g \vee h_1 \vee h_2).$$

For each NOT-gate g of C taking as input gate h we add a set of clauses to f logically equivalent to the statement $g \Leftrightarrow \neg h$, namely the clauses:

$$(g \vee h) \wedge (\neg g \vee \neg h).$$

For each COPY-gate g of C taking as input gate h we add a set of clauses to f logically equivalent to the statement $g \Leftrightarrow h$, namely the clauses:

$$(g \vee \neg h) \wedge (\neg g \vee h).$$

For each constant gate g of C labeled 0 we add a clause $(\neg g)$.

For each constant gate g of C labeled 1 we add a clause (g).

Finally, for the unique output gate g of C we add a clause (g).

The function f is the conjunction of all the clauses described above. Appealing to the Polynomial Church-Turing thesis, the reduction r is seen to be polynomial time computable. Also, we have that C is satisfiable if and only if f is satisfiable: Any truth assignment to the variables of f that makes all clauses `true` corresponds exactly to an evaluation of the circuit C making the output gate `true` and vice versa. This completes the proof of Theorem 12.

Having now established Cook's theorem, we have bootstrapped our database of known **NP**-complete problems: It now contains the problem SAT (and its generalisation CIRCUIT SAT). In the weeks to come, we shall expand it by constructing a tree of reductions to other problems with SAT being the root.

The fact that SAT is **NP**-complete is also a statement which is interesting in its own right and in fact somewhat philosophically intriguing. That a

Boolean formula f is unsatisfiable is equivalent to saying that its negation $\neg f$ is true for *all* truth assignments, i.e., that it is a theorem of propositional logic. For instance, $X \vee \neg X$ is a theorem of propositional logic because $\neg(X \vee \neg X)$ (which is equivalent to the CNF-formula $(\neg X) \wedge (X)$) is not satisfiable. Thus, an algorithm for SAT can be regarded as an automatization of a very special kind of mathematical proof finding activity, namely the activity of refuting statements of propositional logic (of a special form) by finding counterexamples. But recall that MATH, as we defined it in the proof of Proposition 2 is in **NP** and since SAT is **NP**-complete, we have that MATH $\leq$ SAT. Thus, if we want to completely automatize mathematical proof finding *in general*, it is enough to automatize refuting statements of propositional logic by finding counterexamples. In other words, the latter activity is as hard as the former. This is quite a surprising statement!

7 Exercises

Exercise 1 Recall that the Euclidean Traveling Salesman Problem is the special case of the TSP where the distances are actual Euclidean distances between pairs of points. How can we formalize this special case as a language (in the sense of Section 1)? Is the formalization of the special case a special case of the general formalization? (and yes, this question does make sense....)

Exercise 2 Graphs can be represented by adjacency matrices or edge lists. Show how to formally represent adjacency matrices and edge lists as strings and formally define languages corresponding to some decision problems concerning graphs you have previously encountered, such as "Given a directed a graph G and two vertices s and t, is there a path from s to t in G?". In the exercises in the notes on integer linear programming, two optimization problems concerning graphs were defined, namely the maximum independent set problem and the minimum vertex coloring problems. Formally define the two associated decision problems.

Exercise 3 Formally define the languages L_{ILP} and L_{MILP} associated to Integer Linear Programming and Mixed Integer Linear Programming. Show that $L_{\text{TSP}} \leq L_{\text{ILP}}$.

Exercise 4 Prove Proposition 1.

Exercise 5 Prove that the adjacency matrix and edge list representations of graphs, formalized in Exercise 2 are good and polynomially equivalent. Also prove that binary and k-ary, $k > 2$, notation of integers are good and polynomially equivalent. Prove that binary and unary notation of integers

are good but *not* polynomially equivalent.

Exercise 6 Show that the languages associated to the maximum independent set problem and the minimum vertex coloring problems (Exercise 2) are in **NP**.

Exercise 7 Can you prove that the languages L_{ILP} and L_{MILP} of Exercise 3 are in **NP**? If not, what is the obstacle? Let *Bounded* (M)ILP, be the special case of (M)ILP where each variable x_i *must* have two associated constraints $x_i \leq u_i$ and $x_i \geq l_i$ for some integer constants l_i, u_i and let L_{BILP} and L_{BMILP} be the associated languages. Show that L_{BILP} and L_{BMILP} are in **NP**. Actually, is is also true that L_{ILP} and L_{MILP} are in **NP** but the proof is somewhat technical.

Exercise 8 Is it easy to see if the language associated to the Euclidean Traveling Salesman problem of Exercise 1 is in **NP**? Why not?

Exercise 9 Show that L_{BILP} (Exercise 6) is **NP**-complete by a reduction from SAT.