An Overview of Computational Complexity

Ted Rubin

Nov. 18, 2010

1 A Motivation for Complexity Theory

We commonly speak about computers as “problem-solving machines” capable of performing thousands or even millions of operations in a second, and therefore, in terms of time efficiency, being a world apart from humans. In many settings, this characterization is more or less accurate, but there are two reasons, intricately related and equally important, why we should not let it be our final word on the subject. First, computers require programs in order to run, and those programs must be written by humans. Of course, these programs are no more than algorithmic solutions to problems that have been translated, one way or another, into a code that a machine can read. Thus, to take best advantage of the great speed that computers promise, humans somewhere along the line must have a complete understanding of the problem and the method of solution. Second, algorithmic solutions themselves are possessed of some measure of efficiency, which varies even within solutions to the same problem. Since humans may be more or less skilled at writing algorithms, this fact should come as no surprise, but it is also true that the specific problem to be solved has a bearing on the maximum possible efficiency of an algorithmic solution.

Complexity theory is the study of problems and algorithms. In the strictest sense, it can be divorced from computers entirely, but the existence of the digital computer gives us some extremely useful common ground from which to speak about algorithmic problems, because it is for the sake of digital computing that the vast majority of algorithms are developed. When a large computing project is undertaken, efficiency is a major concern. What complexity theory provides is an intuitive measurement scheme for an algorithm’s efficiency, but beyond that, it yields results that can tell us that a problem simply does not have an efficient solution. While not exactly a cause for great rejoicing in applied settings, such results can prevent us from wasting resources looking for a solution that we’ll never find, and cause us to think more carefully about ways in which we can safely restrict the scope of the problem such that an efficient solution might become possible.

1.1 A Problem That (We Suspect) Lacks an Efficient Solution: Boolean Satisfiability

In general, the satisfiability problem asks: given a logical formula using only Boolean variables and the operators AND, OR, and NOT, is there a way to assign values to the variables such that the whole formula evaluates to 'true'?

CNF stands for "conjunctive normal form," which specifies that the variables must be grouped in a way such that the entire formula is a conjunction - clauses connected by AND - of disjuncts, which are individual variables connected by OR. We use the term **literal** to denote either a variable or that variable's negation, so that we can avoid cluttering up our notation with NOTs. Thus we can express the general conjunctive normal form more compactly as

$$(b_{1,1}\vee b_{1,2}\vee ...\vee b_{1,l_1})\wedge(b_{2,1}\vee b_{2,2}\vee ...\vee b_{2,l_2})\wedge ...\wedge(b_{k,1}\vee b_{k,2}\vee ...\vee b_{k,l_k}) = \bigwedge_{i=1}^{k}\bigvee_{j=1}^{l_i} b_{i,j}$$

where k is the number of clauses, l_i for $i = 1, ..., k$ is the number of literals in the ith clause, and $b_{i,j}$ is the jth literal in the ith clause. Because each clause must be true on its own in order for the whole expression to be true, CNF is often easier to work with than some arbitrary Boolean formula. Fortunately, it can be shown that any Boolean expression can be equivalently written in conjunctive normal form.

Suppose we have a Boolean expression F with r variables $a_1, a_2, ..., a_r$ We want to show that F is equivalent to an expression of the form

$$G = \bigwedge_{i=1}^{k}\bigvee_{j=1}^{l_i} b_{i,j}$$

where the $b_{i,j}$s are literals - the positive or negative forms of the a_i's. By "equivalent" we mean that F is true exactly when G is true.

Let Θ be the set of all 2^r possible assignments of F's variables. Then for any assignment $\theta \in \Theta$, and for $1 \leq j \leq t$, we define

$$\alpha_{\theta,j} = \begin{cases} a_j & \text{if } \theta \text{ sets } a_j \text{ to be true} \\ \bar{a}_j & \text{otherwise} \end{cases}$$

We have now tailor-made a set of variables such that $\bigwedge_{j=1}^{t} \alpha_{\theta,j}$ is only satisfied by the assignment θ. Let $S \subseteq \Theta$ be the set of all assignments that satisfy F, and assume that S has k elements. Then

$$F' = \bigwedge_{j=1}^{t} \alpha_{\theta_1,j} \vee \bigwedge_{j=1}^{t} \alpha_{\theta_2,j} \vee ... \vee \bigwedge_{j=1}^{t} \alpha_{\theta_k,j} = \bigvee_{\theta\in S}\bigwedge_{j=1}^{t} \alpha_{\theta,j}$$

is satisfied exactly when one of its disjuncts is satisfied. But this can only happen for an assignment $\theta \in S$, so F' is satisfied exactly when F is satisfied - the two are equivalent.

Using the same method, we can now state that $\bar{F}$ (not-F) is equivalent to an expression of the form

$$\bigvee_i \bigwedge_j b_{i,j}$$

By DeMorgan's laws, F is then equivalent to

$$\bigwedge_i \bigvee_j \bar{b}_{i,j}$$

From now on, we can use the fact that for any Boolean expression, there exists an equivalent Boolean CNF expression.[1]

Developing an algorithm that returns a correct answer for every possible instance of the CNF-Sat problem seems like a daunting task. Many other famous mathematical problems, such as the traveling salesman problem or the subset-sum problem, defy us to come up with a solution procedure that works every time. We might call these problems "hard." The study of complexity theory starts with an attempt to refine this notion of hardness, both in absolute terms - how long does it take to solve the problem? - and in relative terms - which problems can be considered harder than others?

2 Some Definitions

In order to talk about classes of algorithmic problems, we need to have a model of computation that is fairly intuitive and easy to work with. The standard model in complexity theory is the **Turing machine** (TM), a theoretical computing device that has three parts:

- A **tape** that is infinitely long in one direction, and is divided into cells. Each cell can hold one symbol, and in general some of the cells will be written on before computation begins; those symbols are the TM's input. The symbols that can be used by a TM can be chosen for our convenience based on the particular problem we're talking about, but the set of symbols must be finite, and we call it Σ, the TM's **alphabet**. The only other restriction we place on Σ is that is must contain the **blank symbol** σ_0.

  A further note about Σ: we define Σ^* to be the set of all sequences that can be made using the symbols in Σ. This concept is useful when, for example, we want to talk about every possible input that a given TM could receive.

- A **head** that can read the symbols printed on the tape and can also erase them and write new ones. The head and the tape together serve much the same function as the hard drive in a modern computer. The TM head can only operate on one tape cell at a time, so it is also capable of rolling the tape left and right to reach other cells.

[1]Martin pp. 510-12

- A **register** that can be in any of a finite number of states. This is the most abstract part of the model, but it can be understood to function as the memory of the TM. The set of possible states of a TM is called Q. Q always contains a starting state q_0. Since the problems we'll be concerned with are all decision problems, requiring a yes-or-no answer, we'll assume that for all TMs, Q also contains two **halting states**, q_{accept} and q_{reject}. If all goes well, the only way that computation can end is with the TM in one of the two halting states, thus giving us either an affirmative or negative answer.

A TM runs step by step through a program, which can be formally reproduced as a **transition function**

$$\delta := Q \times \Sigma \to Q \times \Sigma \times \{L, S, R\}.$$

In other words, when a new step of computation begins, the TM's register is in some state $q \in Q$, and its head is reading some symbol $\sigma \in \Sigma$ from the tape. On the basis of those two things, the register will change to a new state $q' \in Q$, and the head will overwrite σ with a new symbol $\sigma' \in \Sigma$, then move the tape Left or Right, or Stay reading the same cell. Note that it is perfectly valid for the register to remain in state q, and for the head to avoid actually changing the tape by simply overwriting σ with σ.[2]

Based on our experience with real-life computers, our intuition tells us (correctly) that a TM with the above description operates deterministically. Given the same input more than once, not only will the result of the computation will be the same every time, but the intermediate steps will be exactly the same and traversed in exactly the same order. This kind of consistency is often touted as an advantage that computers have over humans, but in examining complexity classes we quickly come across problems where we might desire a TM that is a bit less consistent.

A **nondeterministic TM** is defined in exactly the same way as a deterministic TM, except that at each step of computation (in other words, when evaluating $\delta(q, \sigma)$), the TM has the potential to transition into a number of different states, write one of a number of symbols to the tape, and move in more than one way. Writing a transition function for a nondeterministic TM is more awkward than it is for a deterministic one. Instead of the nice form above we must write

$$\delta := Q \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{L, S, R\})$$

to indicate that when going from one step to the next the nondeterministic TM selects from a subset of the possible combinations of register states, symbols, and tape movements. Note that the fact that the range is a power set means that $\emptyset$ might be "returned" by δ - that is, δ need not be defined for every pair (q,σ). If a TM encounters such a pair, it will simply be stuck. In practice, such crashes need not worry us as long as we construct our algorithms carefully.

[2]Wegener pp. 22

Because of the cumbersome notation we prefer to specify a nondeterministic TM T using the following 4-tuple:

$$T = (Q, \Sigma, q_0, \delta).$$

We don't know for sure what a nondeterministic TM will do at each step, we know only the set of possible things it could do. To make the notion a bit more concrete, nondeterminism often manifests itself as the ability of the TM to "guess," rather than follow a painstakingly specified algorithm.

Note that a nondeterministic TM is no more capable than a deterministic one. This claim is easily justified by noting that we could program a deterministic TM to run through, in some order that we would specify, every single computation that it is capable of. Even though we can't predict the configurations that a nondeterministic TM will go through in its computation, we can be assured that if we run through all of them using a deterministic TM, we will eventually obtain the same result. Of course, executing this exhaustive simulation would in most cases take an extremely long time, but the salient point is that there is no problem solvable by a nondeterministic TM that is not also solvable by its deterministic counterpart.[3]

In fact, according to **Church's Thesis**, the TM model can be made to suit any algorithm we could possibly come up with. This is an extremely important point, because it means that no matter how fast or capacious real-life digital computers become, they will never be capable of solving an algorithmic problem that a TM could not also be programmed to solve. Church's Thesis is not rigorously provable, but evidence in its favor abounds, and I will make use of it in this paper for the sake of space. Were I not to rely on Church's Thesis, I would have to show explicitly that each algorithm I use can be translated into a TM algorithm, and such proofs would be tedious and uninteresting. As I was invited by every book I used for this paper, I invite my reader to make this one leap of faith and trust that the TM model is not only adequate for this paper, but is the only model that the field of complexity theory should ever need.

2.1 Big-O Notation

We want a classification of algorithmic problems according to the amount of time each one takes to solve. We now have a standardized notion of time: TM computation steps. But a problem remains that will become clear by way of an example.

Suppose we use the following algorithm to determine if a given natural number y is prime: define $z = 2$. If $\frac{y}{z}$ has remainder 0, then halt computation. Otherwise, add 1 to z. Repeat this process until the division produces a zero remainder, in which case we conclude that y is composite, or until z exceeds $\sqrt{y}$, in which case we conclude that y is prime. Church's Thesis assures us that

[3]Homer & Selman pp. 31-3

there exists a TM that can perform this computation, and it's easy to believe that this algorithm will always yield the correct decision - after all, it checks every possible case before concluding that y is prime. But how many steps will the TM take to reach a halting state? Clearly, that depends on the magnitude of y.

In fact, we will never respond to the question "how many steps will the algorithm take?" with a number. Instead, we answer with a **time-complexity function** τ, written in terms of the size of the TM input. Unfortunately, comparing functions directly can get messy. If we have two problems, A and B, we want to be able to make a conclusion about them that is stronger than "$\tau_A(n) > \tau_B(n)$ for some input sizes n, but not all of them." To achieve this goal, we compare bounds.

Suppose we encounter an algorithmic solution to a problem such that $\tau(n) = 5n^3 + 29n^2 + 17n + 4$, where n is the size of the input. In this case we say that $\tau(n) = O(n^3)$. This statement, written in "big-O" notation, formally states that there exist constants C and n_0 such that for all $n > n_0$, $\tau(n) \leq Cn^3$. In other words, as we let n get large, the cubic term of $\tau(n)$ will grow at a much faster rate than the other terms. It grows so much faster that for our purposes we can forget about the other terms, and merely observe that there must be a constant C such that Cn^3 grows faster than $5n^3$. This is how we use big-O notation to express bounds for time-complexity functions.

We've already seen how big-O notation disregards all but the most "powerful" term of a time-complexity function, so all such functions that are cubic, for example, will be indistinguishable from each other. In fact, the classification system used in complexity theory doesn't distinguish among polynomial runtimes of any degree. All algorithms that can be executed by a TM with polynomial worst-case runtime fall into one complexity class. In practice we are only interested in two other broad classes of algorithms: those that run with subpolynomial worst-case runtime (for example, $O(\text{log}n)$); and those that run with exponential worst-case runtime (for example, $O(c^n)$ for some constant c), and are associated with intractable problems.

2.2 Two Important Complexity Classes

The most famous complexity classes are P and NP; in fact the question of whether or not P = NP is the most important open problem in the field of complexity theory. These two classes contain the problems that are most interesting and accessible to us with our current resources - that is to say, with digital computers.

The class P, roughly speaking, contains all problems that we consider to be efficiently solvable. Any problem that can be solved by a deterministic TM with polynomial worst-case runtime is in P. Thus, a proof that a problem is in P involves finding an algorithm for solving that problem that could be executed by a deterministic TM in polynomial time.

The class NP can be thought of as containing all problems with solutions that can be easily verified. The CNF-Sat problem, for example, is in NP because it is easy to check whether a given variable assignment makes a Boolean expression evaluate to "true." More formally, a problem is in NP if there exists a nondeterministic TM that can solve it with polynomial worst-case runtime (in fact, NP stands for "nondeterministic polynomial"). An equivalent definition of membership in NP is that there exists a deterministic TM which, given a solution to the problem, can verify that the solution works, again in polynomial time.[4]

We said above that nondeterministic TMs have the ability to "guess" while solving problems, and this is an appropriate characterization because, unlike for problems in P, we do not have any algorithm for NP problems that is more efficient than guessing and checking. Herein lies the importance of the $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ problem: if in fact $\mathrm{P} = \mathrm{NP}$ then we can be assured that there exist efficient algorithmic solutions to problems like CNF-Sat, the Traveling Salesman problem, and integer factorization. That being said, we strongly suspect that $\mathrm{P} \neq \mathrm{NP}$, but so far attempts at proof have been unsuccessful. Note, however, that P is surely a subset of NP, because a nondeterministic TM can also solve any problem that a deterministic TM can.

Two subclasses bear mentioning at this point. A problem is called **NP-hard** if it is at least as hard as every problem in NP. Further, an NP-hard problem is called **NP-complete** if the problem is itself in NP.[5]

2.3 Reductions

It should come as no surprise that proving that a problem is at least as hard as every other problem in NP is itself a rather difficult task. Fortunately we are assisted by polynomial-time Turing reductions, which we will refer to simply as Turing reductions from now on because reductions that take longer than polynomial time are not of interest to us here. Informally, a Turing reduction is the process of mapping one problem onto another. In order to be more specific, we introduce the concept of an **oracle**. An oracle is simply an algorithmic "black box" for solving a problem. Such a thing can exist only in theory, but we use it to ask the question "can we transform an instance of problem B into one or more instances of problem A, which, if we assume the existence of an A-oracle, we would then be able to solve?"

So, to perform a reduction, we write an algorithm for B that runs in polynomial time with respect to the size of the input. We allow the algorithm to call for the assistance of an A-oracle, as long as the number of calls it makes is polynomially bounded. Every time the algorithm calls the A-oracle, it will have to provide input, since the A-oracle is itself an algorithm. Thus we also require that the size of the input in each call to the A-oracle is polynomially bounded.[6]

If such an algorithm can be constructed, we say that B is reducible to A. We will see several reduction examples after proving our main result.

[4] Arora & Barak pp. 39
[5] Martin pp. 509-10
[6] Wegener pp. 44-5

3 Cook's Theorem

While Turing reductions can save us quite a bit of work when classifying problems, we still need to do some heavy lifting to prove that an NP-complete problem exists at all. Specifically, our goal is to prove that the Boolean satisfiability problem (Sat) is NP-complete. Instead of doing so directly, however, we'll instead go through a proof that CNF-Sat is NP-complete, and then use the method of Turing reduction to conclude that Sat is NP-complete.

To prove NP-completeness we must first prove that CNF-Sat is in NP. We assume that we have a CNF expression F and a variable assignment θ. What we need to show is that a TM can verify whether θ satisfies F in polynomial time, once F has been encoded on the TM's tape. At this point we should be specific about the alphabet being used. Define

$$\Sigma_{CNF} = \{\wedge, w, \bar{w}, 1, \sigma_0\}.$$

Our use of this alphabet will be made clear through an example. We encode the Boolean expression

$$(w_1 \vee w_2) \wedge (\bar{w}_2 \vee w_3 \vee w_4)$$

as

$$w1w11 \wedge \bar{w}11w111w1111$$

with the bar notation indicating negation. We can omit a symbol for OR because the literals within a clause are all joined by OR, so we only need to separate the clauses with $\wedge$. The use of unary notation instead of a more compact subscript-writing method will simplify the question of how long the TM will take to read the variables in F.

The TM follows a simple algorithm: starting at the beginning of a clause (reading w or $\bar{w}$), it identifies the first literal by reading until it reaches a non-1 symbol. If θ has made that literal true, then the TM reads over the following symbols until it reaches a $\wedge$, which marks the beginning of the next clause, at which point it repeats the procedure. If the identified literal is not true under θ, then the TM identifies and checks the next literal in the clause. If the TM reaches the end of a clause without finding a true literal, it halts computation in q_{reject}, having found that θ does not satisfy F. Otherwise, the TM will eventually reach a blank cell, in which case computation ends in q_{accept} (unless it found that none of the literals in the final clause were true).

Using this algorithm, the TM head only needs to make one pass over the formula encoded on the tape. Since we express time complexity in terms of the size N of the input, we then say that this verification algorithm runs in $O(N)$ time, which places CNF-Sat comfortably in NP.

Now we need to prove that CNF-Sat is NP-hard. We observe that for every problem in NP there exists a nondeterministic TM $T = (Q, \Sigma, q_0, \delta)$ that solves it in polynomial time. Fix T, and choose a polynomial $p(n)$ such that $\tau_T(n) \leq p(n)$ for all n, where n is defined to be $|x|$, the size of T's input. We also define $N = p(n)$, our upper bound on the time complexity of T.

Giving the name CNF to the set of all Boolean CNF expressions, we want to show that there exists a function

$$g : \Sigma^* \to \text{CNF},$$

which takes TM inputs and returns Boolean expressions, such that the following two conditions are fulfilled.

- For any $x \in \Sigma^*$ that is the input for T, T halts in an accepting state if and only if $g(x)$ is a satisfiable Boolean expression.
- The reduction of the problem computed by T to the problem CNF-Sat can be performed in polynomial time.

We define a **configuration** of a nondeterministic TM to be a 3-tuple $(q, \sigma, n) \in Q \times \Sigma \times \mathbb{N}$, representing a computation step in which the TM is in state q reading the symbol σ from the nth cell of the tape. We then observe that, even though we cannot make predictions about the computation steps of a nondeterministic TM, we can view its problem-solving process as a sequence of configurations. Further, we can make judgments about whether such a sequence, or **computation path**, is valid or not by examining the set of possible transitions at each step of computation. We introduce the relation $\vdash$ on the set of configurations of a TM. $C \vdash C'$ implies that a TM can go from configuration C to configuration C' in one step. We then call C and C' **consecutive**.

We can now define a valid computation path as one that begins with $C_0 = (q_0, \sigma, 0)$, and in which, for every configuration C_i, $C_i \vdash C_{i+1}$. The notion of a valid computation path gives us a goal to work for in constructing g: we desire that for any input x, $g(x)$ is satisfiable if and only if there is a valid computation path from $C_0 = (q_0, x_0, 0)$ to a configuration in which T is the state q_{accept}. The easiest way to construct such a g will be to force the CNF expression that it produces to be an expression that we can associate with the TM itself.

We first require an indexing for all the states and symbols that could be used by T. We'll call the states

$$Q = \{q_0, q_1, ..., q_{t-1}, q_t\}$$

where t is the total number of states, q_0 is the starting state, as before, and q_{t-1} and q_t are how we refer to the rejecting and accepting states, respectively. Similarly we'll call the symbols

$$\Sigma = \{\sigma_0, \sigma_1, ..., \sigma_s\}$$

where $s + 1$ is the number of symbols used by the TM, and σ_0 is the blank symbol. Using this notation we express the starting state of the TM's tape - that is, its input - as $x = \sigma_{a_1}\sigma_{a_2}...\sigma_{a_n}$.

To represent something as complicated as a TM using something as simple as a Boolean formula will require a great many variables. We can divide the variables we will require into three groups.

- For $i = 1, ..., N$ and $j = 0, ..., t$, the variable $Q_{i,j}$ is true if and only if, after the ith step, T is in state q_j.
- For $i = 1, ..., N$ and $k = 0, ..., N$, the variable $H_{i,k}$ is true if and only if, after the ith step, the tape head is reading the kth cell of the tape. Note that it is possible for us to bound k because it is impossible for the TM head, starting at cell 0, to go past cell N in N moves.
- For $i = 1, ..., N$ and $k = 0, ..., N$, and $l = 0, ..., s$, the variable $S_{i,k,l}$ is true if and only if, after the ith step, the symbol σ_l is printed in tape cell k.

We can now proceed to construct $g(x)$, a conjunction of seven expressions. As we examine the expressions in turn, we will verify that each one includes a number of variables that is polynomially bounded. After examining each expression we will conclude that encoding $g(x)$ in the language of the TM will require a number of cells that is still polynomially bounded.

1. $$Q_{0,0} \wedge H_{0,0} \wedge S_{0,0,0} \wedge \bigwedge_{k=1}^{n} S_{0,k,a_k} \wedge \bigwedge_{k=n+1}^{N} S_{0,k,0}$$

   The first expression represents the initial configuration of the TM. If this expression is true, then all the following statements are true: After 0 moves, the TM is in state q_0. The tape head is reading from cell 0, which contains the blank symbol. Every cell k from the 1st to the nth contains the symbol σ_{a_k}, and every cell from the $n+1$st to the Nth contains the blank symbol.

   This expression requires $3 + N$ variables.

2. $$Q_{N,t}$$

   This much simpler expression is true if and only if, after the Nth step, T is in the accepting state.

3. $$\bigwedge_{i=0}^{N-1} \bigwedge_{k=0}^{i} \bigwedge_{j,l} \left((Q_{i,j} \wedge H_{i,k} \wedge S_{i,k,l}) \rightarrow \bigvee_{m} (Q_{i+1,j_m} \wedge H_{i+1,k_m} \wedge S_{i+1,k,l_m}) \right)$$

   $$\text{for } m = 0, ..., M$$

   $$(j,l) \text{ varies over all pairs such that } \delta(q_j, \sigma_l) \neq \emptyset$$

   This expression represents the transition function $\delta(q_j, \sigma_l)$ of T. In order to better understand what the expression is saying, fix i, j, k, and l. The left side of the implication specifies a configuration of T after step i. Then, on move $i+1$, T will transition into one of M configurations, where M is constant for each configuration. The value of M depends on i, j, k, and l. In other words, each of T's configurations might have a different number of consecutive configurations.

Note that k_m can only take on values $k-1, k$, or $k+1$, because the head position can shift by at most one cell in one step.

One might reasonably object that the Boolean OR on the right side allows more than one configuration to be "true." Expressions 5 and 6 will patch this particular hole, since T will enter exactly one consecutive configuration, even though we cannot predict which configuration it will be.

We'll count the number of variables in this expression together with those in Expression 4.

4.
$$\bigwedge_{i=0}^{N-1} \bigwedge_{k=0}^{i} \bigwedge_{j,l} ((Q_{i,j} \wedge H_{i,k} \wedge S_{i,k,l}) \rightarrow (Q_{i+1,j} \wedge H_{i+1,k} \wedge S_{i+1,k,l}))$$

$$(j,l) \text{ varies over all pairs for which } \delta(q_j, \sigma_l) = \emptyset$$

This expression serves as the complement to Expression 3, since it addresses all those pairs (j,l) for which Expression 3 was not defined. If after the ith step T is in a configuration C such that its transition function evaluates to the empty set, we state equivalently that C has no consecutive configurations. The machine has essentially crashed, and the configuration remains unchanged after the $i+1$st step. In fact, T will be unable to change its configuration ever again (though this expression need not reflect that).

We can count the number of variables for Expressions 3 and 4 together, after taking care of a small difficulty. "$\rightarrow$" is not a symbol in our CNF-Sat language, and was included in these two expressions only in order to make their purpose clear. Thus when the TM is actually writing out $g(x)$, each instance of the implication in Expression 3 should be written in the logically equivalent form

$$\left(\bar{Q}_{i,j} \vee \bar{H}_{i,k} \vee \bar{S}_{i,k,l} \vee Q_{i+1,j_m}\right)$$
$$\wedge \left(\bar{Q}_{i,j} \vee \bar{H}_{i,k} \vee \bar{S}_{i,k,l} \vee H_{i+1,k_m}\right)$$
$$\wedge \left(\bar{Q}_{i,j} \vee \bar{H}_{i,k} \vee \bar{S}_{i,k,l} \vee S_{i+1,k,l_m}\right).[7]$$

A similar transformation can be performed for the implication of Expression 4, except that the subscript m is not necessary. Note that the two expressions are in proper conjunctive normal form, and that our "transformation" does not affect the polynomial time complexity of $g(x)$.

[7]Homer & Selman pp. 135

There are st total pairs (j, l), each of which is accounted for exactly once, either in Expression 3 or in Expression 4. i takes on one of N possible values in each expression, and k takes on a number of values that is bounded by N. Thus the number of times we need to write each expression is bounded by stN^2. Fortunately, s and t are constant for a given TM, and M for each configuration is constant as well, so the number of times we need to write each expression is $O(N^2)$.

5.
$$\bigwedge_{i=1}^{N} \bigvee_{j=0}^{t} \left(Q_{i,j} \wedge \bigwedge_{h \neq j} \bar{Q}_{i,h} \right)$$

With this expression we begin to address one of the shortcomings of Expression 3, and enforce a vital rule of TMs. At every step, T is in exactly one state, or to be more specific, T is in a state q_j, and for all $h \neq j$, T is not in the state q_h.

i takes on one of N values, and j takes on one of $t + 1$ values, which is a constant for a given TM. So each $Q_{i,j}$ must be written tN times, and the each $\bar{Q}_{i,h}$ must be written $t(t-1)N$ times. Both of these are $O(N)$.

6.
$$\bigwedge_{i=0}^{N} \bigwedge_{k=0}^{i} \left[\left(\bigvee_{l} S_{i,k,l} \right) \wedge \left(\bigwedge_{l_1,l_2} (\bar{S}_{i,k,l_1} \vee \bar{S}_{i,k,l_2}) \right) \right]$$
$$\text{for } (l_1, l_2) \text{ such that } l_1 \neq l_2$$

Here we remedy another source of configurations that would cause the expressions so far to evaluate true, but would in fact be completely illegal by TM rules. At every step, and for all $k = 0, ..., N$, the kth cell contains at least one symbol, and never contains two symbols.

Each $S_{i,k,l}$ must be written a number of times bounded by sN^2, and each $\bar{S}_{i,k,l_1} \vee \bar{S}_{i,k,l_2}$ must be written a number of times bounded by $s(s-1)N^2$. Both of these are $O(N^2)$ because s, the size of the set Σ, is constant for a given TM.

7.
$$\bigwedge_{i=0}^{N} \bigwedge_{k=0}^{i} \bigwedge_{l=0}^{s} (\bar{H}_{i,k} \wedge S_{i,k,l} \rightarrow S_{i+1,k,l})$$

Finally, the TM can only change the cell being read by the head. Therefore for any step i and any cell k containing the symbol σ_l, if the head is not reading cell k after the ith step then cell k will still contain σ_l after the $i + 1$st step.

Again, T cannot actually understand "$\rightarrow$", so it will instead write the logically equivalent clause

$$H_{i,k} \vee \bar{S}_{i,k,l} \vee S_{i+1,k,l}$$

This clause must be written a number of times bounded by sN^2, which is $O(N^2)$.

We now have seven expressions that serve as our conjuncts to form $\bigwedge_{i=1}^{7} g_i(x) = g(x)$. $g(x)$ will be true if and only if there exists a valid computation path from C_0 to a configuration that has T in the accepting state. Further, we have a polynomial bound on the amount of time it will take to write out each part of $g(x)$. Since the sum of polynomials is itself a polynomial with degree no higher than the degrees of the summands, we can say that writing out all of $g(x)$ has time complexity $O(N^2)$.[8]

4 The Fruit of Cook's Theorem: Some Turing Reductions

Now that we have a problem that is surely NP-complete - that is, contained in the class of the hardest NP problems - the task of proving that certain other problems in NP are also NP-complete is much easier. We merely need to show the existence of a Turing reduction from CNF-Sat to another NP problem, meaning that CNF-Sat is no more difficult than that problem, implying that the other problem is also NP-complete.

4.1 Reduction of CNF-Sat to Sat

Though the restricted forms of Sat are most often nicer to work with, we would still like to have a proof that the most general form of the problem is NP-complete. We will show that Sat accords with our second definition of membership in NP by assuming that we have both a Boolean expression f and an assignment θ that satisfies it, and then proving (by providing an algorithm) that a deterministic TM could verify the satisfying assignment in polynomial time.

The first step in the algorithm is to run through the formula once, replacing all the literals made true by θ with a 1 and all the ones made false with a 0. This step only needs to be performed once and takes $O(n)$ time. The next step is to find pairs of digits connected by operators and replace them with a single digit, according to the rules of logic. For example,

$$0 \wedge 1 \text{ becomes } 0$$

$$1 \wedge 1 \text{ becomes } 1$$

$$1 \vee 0 \text{ becomes } 1$$

[8] Martin pp. 512-17

This step is complete when there is only one digit left, at which point we decide that θ satisfies f if and only if that lone digit is a 1. Note that every time this operation is performed the total number of digits on the tape is reduced by 1, thus it must be performed $n-1$ times. Thus the total time complexity of the verification algorithm is $O(n)$ and Sat is in NP.

At this point we do not actually need a formal reduction to show the NP-completeness of Sat. Rather, it will suffice to observe that because every CNF-Sat problem is also a Sat problem, CNF-Sat is a restriction of Sat. Thus Sat can be no less complex than CNF-Sat. By showing that Sat is in NP, we have shown that it is also no more complex than CNF-Sat. It then follows that Sat is also NP-complete.

Note that this result implies that there exists a polynomial-time algorithm that will transform any Sat expression into CNF-Sat. Earlier, we showed a transformation, but because it depended on an enumeration of all possible truth assignments for a Boolean formula, it ran in exponential time. Because both problems are NP-complete, we now know that a more efficient transformation must exist.

4.2 Reduction of CNF-Sat to 3-Sat

We perform this reduction mainly for the sake of following Turing reductions. 3-Sat is a special case of CNF-Sat where each clause contains exactly three literals. Thus we need to transform an arbitrary CNF expression F into a 3-Sat expression G.

Obviously, if a clause of F already has three literals, we can leave it alone. For clauses with one or two literals, we repeat a literal within the clause to bring it up to three, which does not change that logical properties of F at all. The more involved case is when a clause c has $k > 3$ literals:

$$c = w_1 \vee w_2 \vee w_3 \vee ... \vee w_k$$

We observe that c is satisfied if even one of the w_is is true. We need to split the variables of c into new clauses in such a way that the conjunction as a whole retains this property. The splitting will involve creating a new set of variables $\{y_1, y_2, ..., y_{k-3}\}$. We then create

$$d = (w_1 \vee w_2 \vee y_1) \wedge (\bar{y}_1 \vee w_3 \vee y_2) \wedge (\bar{y}_2 \vee w_4 \vee y_3) \wedge ... \wedge (\bar{y}_{k-4} \vee w_{k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee w_{k-1} \vee w_k).$$

We need to show that c is satisfied if and only if we can assign truth values to the y_is such that d is satisfied. After confirming this, we can safely proceed to replace every clause of F with a number of 3-Sat clauses, for the purpose of submitting the entire expression to a 3-Sat oracle.

($\Rightarrow$) Assume that c has a satisfying assignment. Since c is a disjunction of literals, we know that one of the w_is must be true - call one of them w_T. For all $i \leq T - 2$ (that is, in every clause to the left of the one containing w_T), we let y_i be true, and for all the rest we let y_i be false. In this way every clause of d

up to the one with w_T is surely satisfied. For $i > T-2$, we let y_i be false. Since all of the clauses to the right of the one containing w_T contain the negation of one of these y_is, they are now satisfied as well, so d as a whole is satisfied.

($\Leftarrow$) Assume that we have a variable assignment θ (for both the w_is and the y_is) that satisfies d. In order to show that c must also be satisfied by this assignment, we need to show that one of the w_is is by itself sufficient to satisfy one of the clauses of d. Let j be the smallest possible index value such that θ makes y_j false. Then the clause containing y_j is not satisfied by a y variable, because we stipulated that y_{j-1} is true, making $\bar{y}_{j-1}$, the other variable sharing a clause with y_j, false. But since we assumed that the clause is satisfied, w_{j+1} must be true. This further implies that θ satisfies c.

Once we have made this transformation from a CNF formula to a 3-Sat formula, we need only make one call to a 3-Sat oracle to get an answer, so the number of calls to the oracle is clearly polynomially bounded. Within the newly generated 3-Sat formula, there are not even three times as many literals as there are in the original CNF formula F, so the size of the input to the 3-Sat oracle, in terms of the size of F, is also polynomially bounded. Finally, aside from calling the 3-Sat oracle, the only thing that our CNF-Sat algorithm does is transform the clauses of F one at a time into clauses that fit the form we defined above. Each clause only needs to be considered once so the time complexity of this process is $O(l)$, where l is the number of literals in F. So, all three polynomial bound requirements are satisfied, giving us the reduction CNF-Sat $\leq_T$ 3-Sat.

The reduction gives us a lower bound on the complexity of 3-Sat, but alone is not enough to prove NP-completeness, because we do not yet know that 3-Sat is not any more complex than CNF-Sat, i.e., that it is in NP. Fortunately we can use a trick similar to the one we used in the previous proof - we merely need to observe that 3-Sat is a restriction of CNF-Sat and must therefore be in NP. Thus 3-Sat is NP-complete.[9]

4.3 Reduction of 3-Sat to Clique

Since graph theory appears to be in vogue among senior exercises this year, we would be remiss if we failed to examine some of the many NP-complete problems that are stated in terms of graphs.

First, some definitions. We specify a **graph** G by the pair (V, E), where V is a set of **vertices** and E is a set of **edges**. In turn, an edge is a pair of vertices (v, v'). We then say that an edge is **incident** to the vertices v and v', or that v and v' are **connected**.

A **clique** (also known as a **complete subgraph**) is a set of vertices within which every vertex is connected to every other vertex.

Now we can state the clique problem: given a graph G and an integer k, is there a subset of G that is a clique containing k vertices? What we need to find

[9]Wegener pp. 51-2

is a polynomial-time algorithm for turning an instance of 3-Sat into an instance of the clique problem. Let f be an arbitrary 3-Sat formula.

It seems reasonable to represent the literals of f by vertices in G. Let m be the number of clauses in f - then G will have $3m$ vertices. We will index the vertices such that the vertex $v_{i,j}$ represents the jth literal in the ith clause of f.

Now we need to decide which vertices to connect with edges. To begin with, for all i, the vertices $v_{i,1}, v_{i,2}$, and $v_{i,3}$ will not be connected. Further, we will not connect two vertices if they cannot be true at the same time. Of course, this only happens when the two vertices represent literals that are each others' negations. All other pairs of vertices - that is, those pairs in which the two literals represented are not in the same clause and can be true at the same time - will be connected by an edge. We must now prove that such a graph contains a clique of size m if and only if f is satisfiable.
($\Rightarrow$) Suppose we have constructed a graph G according to the above rules, and G contains a clique of size m. We will denote by V the subset of G's vertices that form the clique. Since we stipulated that vertices representing literals from the same clause are never connected by edges, the clique must contain a representative vertex from every clause of f.

To prove that f is satisfiable, it suffices to produce a satisfying assignment. Let $\theta = (a_{1,1}, a_{1,2}, ..., a_{m-1,3}, a_{m,1}, a_{m,2}, a_{m,3})$ be an assignment to the variables of f such that $a_{i,j} = 1$ if $v_{i,j}$ is in V, and is 0 otherwise. Since V contains one representative vertex from each clause of f, θ makes one variable from each clause of f true. But this is exactly enough for θ to be a satisfying assignment of f, so f must be satisfiable.
($\Leftarrow$) Let θ be a satisfying assignment of the variables in f. Then each clause of f contains at least one literal that has been made true by θ. Since the order of the literals within a clause is irrelevant, assume without loss of generality that θ makes the first literal in each clause true. Then for all $1 \leq i < j \leq m$, the vertices $v_{i,1}$ and $v_{j,1}$ are connected by an edge. We know this because they represent literals from different clauses, and they cannot contradict each other because we were given θ, which made them all simultaneously true. There are m of these vertices, and each is connected to every other, forming a clique of size m.

We do not need to go into detail about how we would create a TM representation of the graph G on a linear tape (and we can appeal to Church's Thesis to assert that such a representation is possible). What is important to observe is that the TM representation of one vertex of G would differ in length from that of another vertex by no more than a constant, just like our variables from Cook's Theorem that were represented by an x followed by a string of 1s. The same can be said of edges, which are simply unordered pairs of vertices. Thus, to verify that the construction of a graph based on f can be performed in polynomial time, we need only note that the graph contains $3m$ vertices and at most $\binom{3m}{2} = O(m^2)$ edges, so the size of the input to a clique-oracle is polynomially bounded. Finally, once we have constructed G, we need only call the clique-oracle once - it will either find a clique or tell us that none exists. Thus

3-Sat is Turing reducible to Clique.[10]

4.4 Reduction of Clique to Vertex Cover

Continuing with graph theory, we examine the vertex cover problem: Given a graph G and a positive integer k, is there a subset S, containing k of G's vertices, such that all edges in G are incident to at least one vertex in S?

A polynomial-time transformation of an instance of the clique problem into an instance of the vertex cover problem proceeds as follows. Let (H, j) be an instance of the clique problem, with the graph $H = (V, E_H)$ and a desired clique size j. Construct a new graph $G = (V, E_G)$ with the same vertices as H, but with precisely the opposite edges - that is, if two vertices are connected in H, they are not connected in G, and if they are not connected in H, there is an edge between them in G. Let $k = |V| - j$, where $|V|$ is the number of vertices in H (and G). We claim that H contains a clique of size j if and only if G contains a vertex cover of size k.

($\Rightarrow$) Assume that $\{v_1, v_2, ..., v_j\}$ is a clique in H. Then none of the v_i's are connected in G as we constructed it. That means that every edge of G must be incident to a vertex in the set $V \setminus \{v_1, v_2, ..., v_j\}$. This set contains $|V| - j = k$ vertices and is a vertex cover of G.

($\Leftarrow$) Assume that $\{u_1, u_2, ..., u_k\}$ is a vertex cover of G. Because every edge in G is incident to a vertex in $\{u_1, u_2, ..., u_k\}$, any pair of vertices that we could choose from $V \setminus \{u_1, u_2, ..., u_k\}$ are not connected by an edge in G. Because of the way we constructed G, this means that every pair of vertices in $V \setminus \{u_1, u_2, ..., u_k\}$ is connected by an edge in H. The set $V \setminus \{u_1, u_2, ..., u_k\}$ has $|V| - k = j$ vertices, and is therefore a clique of size j in H.[11]

To complete the reduction we show that the transformation can be carried out in polynomial time. Starting with the graph H, let $n = |V|$ be our measurement of the size of the input. We list all possible edges connecting vertices in V. Since there are $\binom{n}{2}$ possible edges, this takes $O(n^2)$ time. We then read through the edges of H and delete those edges from our exhaustive list, which takes $O(n)$ time. We are left with the vertices of H (which are also the vertices of G) and the edges of G. Thus the construction of G can be done in polynomial time.

4.5 Reduction of 3-Sat to 3DM

Moving on from graph theory, we approach a more difficult reduction, proving that the 3-dimensional matching problem is NP-complete. 3DM centers around three disjoint finite sets W, X, and Y, each of which contains q elements. We'll call these three sets **dimensions**. We take $R \subseteq W \times X \times Y$, a set of 3-tuples that is fixed in a given instance of 3DM. We are interested in whether or not there exists a subset M of R that is a **matching**. A matching is a set that fulfills the following conditions:

[10]Wegener pp. 56-7
[11]Martin pp. 520

1. It has q elements.
2. No two of its elements have any coordinates in common.

As a quick example, let $W = \{1, 2, 3\}, X = \{A, B, C\}$, and $Y = \{\heartsuit, \clubsuit, \spadesuit\}$. Then $M = \{(1, A, \heartsuit), (2, B, \clubsuit), (3, C, \spadesuit)\}$ is a matching subset of $W \times X \times Y$. This matching isn't hard to find, because we did not impose a restricting set R. If we did so such that, say, $(1, A, \heartsuit) \notin R$, we would have to do a bit more shuffling of the elements of W, X, and Y before finding a matching set, and one can easily imagine that the most interesting instances of 3DM occur when R is very restrictive indeed.

It's not hard to show that 3DM is in NP. Given a proposed matching $M \subseteq R$, we need to check that it has q elements, which can be done in linear time. Then we need to check that none of the coordinates among the elements of M are duplicates. For each dimension, this check requires $\binom{q}{2} = O(q^2)$ pairwise comparisons. Thus, verification of a solution to an instance of 3DM can be performed in polynomial time, which means that 3DM is in NP.

Now we proceed with a polynomial-time transformation from 3-Sat to 3DM. We start with a set of variables $U = \{u_1, u_2, ..., u_n\}$, grouped into a set of clauses $C = \{c_1, c_2, ..., c_m\}$, forming the 3-Sat expression F. First, we have to create the three dimensions. The elements of the dimensions may seem bizarre at first - we will go through the purpose of each of them in turn.

$$W = \{u_{i,j}, \bar{u}_{i,j} : 1 \leq i \leq n, 1 \leq j \leq m\}$$

Note that these $u_{i,j}$'s are *not* the variables of F! There is a $u_{i,j}$ and a $\bar{u}_{i,j}$ for every combination of variables (i's) and clauses (j's) in F, even when a given variable does not appear in a given clause.

$$X = A \cup S_Y \cup G_Y$$

where we set

$$\begin{aligned} A &= \{a_{i,j} : 1 \leq i \leq n, 1 \leq j \leq m\} \\ S_X &= \{s_{X,j} : 1 \leq j \leq m\} \\ G_X &= \{g_{X,k} : 1 \leq k \leq m(n-1)\}. \end{aligned}$$

Similarly,

$$Y = B \cup S_Y \cup G_Y$$

where we set

$$\begin{aligned} B &= \{b_{i,j} : 1 \leq i \leq n, 1 \leq j \leq m\} \\ S_Y &= \{s_{Y,j} : 1 \leq j \leq m\} \\ G_Y &= \{g_{Y,k} : 1 \leq k \leq m(n-1)\}. \end{aligned}$$

Observe that each dimension contains $2mn$ elements, so M, if it exists, will have to contain $2mn$ 3-tuples.

Of course, determining whether or not F is satisfiable only requires knowledge of the variables and clauses of F. The contents of X and Y are completely alien to the 3-Sat problem, but they are essential to the transformation because we want to be able to create the 3-tuples of 3DM without having to mix representations of literals together. In other words, they are there to fill space, albeit in a very systematic way. Choosing among 3-tuples taken from these three dimensions, our goal is to create a restricting set R such that a subset of R that is a matching exists if and only if F is satisfiable. We will construct R explicitly such that the forward direction holds, and then we'll prove the reverse direction.

There will be three different classes of 3-tuples in R, each serving a distinct function. The first class is the T_i's ($1 \leq i \leq n$). The T_i's are further divided into the T_i^f's and the T_i^t's. R will contain both of these subclasses but for a fixed i, the matching M, if it exists, will contain only the T_i^f's if the satisfying truth assignment to F makes u_i false. It will contain only the T_i^t's if the truth assignment makes u_i true.

Further, this class is the only place where we will make use of the $a_{i,j}$'s from A and the $b_{i,j}$'s from B. The purpose of these "dummy elements" is to keep the 3-tuples of the T_i's distinct from each other, as well as to pad them out to 3 elements. The precise structures are:

$$T_i^t = \{(\bar{u}_{i,j}, a_{i,j}, b_{i,j}) : 1 \leq j \leq m\}$$

$$T_i^f = \{(u_{i,j}, a_{i,j+1}, b_{i,j}) : 1 \leq j < m\} \cup \{(u_{i,m}, a_{i,1}, b_{i,m})\}$$

The only important 3-Sat information that is encoded in the T_i's is in the $u_{i,j}$'s and the $\bar{u}_{i,j}$'s, but we must obey 3DM form and create 3-tuples, so we invent the $a_{i,j}$'s and $b_{i,j}$'s. Note that both of these sets contain exactly m 3-tuples, and also that if a certain $a_{p,q}$ and $b_{k,l}$ appear together in one of the T_i's, they will not appear together in any other T_i. This preserves the eligibility of the T_i's as part of a matching.

A word on the encoding that we have performed with the T_i's: it may seem exactly backwards to put the $u_{i,j}$'s into the "u_i is false" subclass T_i^f and the $\bar{u}_{i,j}$'s into the "u_i is true" subclass T_i^t. The reason for this choice will become clear after we construct the next class of 3-tuples.

Now that we have encoded the possible truth values of F's literals, we need to encode the arrangement of the literals among its m clauses. We'll call this class of 3-tuples the C_j's, with $1 \leq j \leq m$. For this class, we use the $s_{X,j}$'s and the $s_{Y,j}$'s to fill space, similar to the way we used the $a_{i,j}$'s and $b_{i,j}$'s before. Each member of this class is of the form

$$C_j = \{(u_{i,j}, s_{X,j}, s_{Y,j}) : u_i \in c_j\} \cup \{(\bar{u}_{i,j}, s_{X,j}, s_{Y,j}) : \bar{u}_i \in c_j\}.$$

We assume without loss of generality that none of the c_j's contain repeated literals (3-Sat allows this to happen when we need to expand a clause of one or two literals to fit the form), so for each j, C_j contains three 3-tuples. To include all of the $s_{X,j}$'s and $s_{Y,j}$'s, as required, M will have to include exactly one 3-tuple from C_j for each j. If a clause c_j of F is satisfied by some u_i, then $T_i^t \subseteq M$, which puts $\bar{u}_{i,j}$ into M, and then we can put $(u_{i,j}, s_{X,j}, s_{Y,j}) \in C_j$ into M as well, so both $\bar{u}_{i,j}$ and $u_{i,j}$ are accounted for exactly once in M, which is what we want. The situation runs similarly when c_j is satisfied by some $\bar{u}_i$.

The final class of 3-tuples makes use of the $g_{X,k}$'s and $g_{Y,k}$'s. Its purpose is to include those $u_{i,j}$'s and $\bar{u}_{i,j}$'s that have so far been excluded from M because of redundancy. Specifically, it is possible that a clause of F contains more than one true literal, but only one satisfying literal per clause can be accounted for in $C_j \cap M$, or else M would be invalid because it would contain duplicate $s_{X,j}$'s and $s_{Y,j}$'s. So G stands for "garbage collection," and takes the form

$$G = \{(u_{i,j}, g_{X,k}, g_{Y,k}), (\bar{u}_{i,j}, g_{X,k}, g_{Y,k}) : 1 \leq k \leq m(n-1), 1 \leq i \leq n, 1 \leq j \leq m\}.$$

Although G contains an exhaustive catalog of the $\bar{u}_{i,j}$'s and $u_{i,j}$'s, only $m(n-1)$ of its 3-tuples will be used by M - that is, enough so that all of the $g_{X,k}$'s and $g_{Y,k}$'s will be used and the total number of 3-tuples in M comes to $2mn$.

We have now constructed our dimensions and specified how we will choose our restricting set:

$$R = \left(\bigcup_{i=1}^{n} T_i\right) \cup \left(\bigcup_{j=1}^{m} C_j\right) \cup G$$

We have seen how the existence of a valid matching using the elements of R implies that a satisfying assignment exists for F. Now we prove the converse. Let θ be an assignment that satisfies F. We will show that a valid matching exists by constructing one. For each clause c_j in F, let z_j be one of the three literals in c_j that is made true by θ, and then let u_{z_j} be the $u_{i,j}$ or $\bar{u}_{i,j}$ corresponding to that literal. Our matching is then

$$M = \left(\bigcup_{u_i \text{true}} T_i^t\right) \cup \left(\bigcup_{u_i \text{false}} T_i^f\right) \cup \left(\bigcup_{j=1}^{m} \{(u_{z_j}, s_{X,j}, s_{Y,j})\}\right) \cup G'$$

where G' is the subset of G containing $m(n-1)$ 3-tuples that we discussed above. We can be assured that the count of 3-tuples can always work out as we expect, because M contains exactly mn 3-tuples from the T_i's, and m 3-tuples from the C_j's, so in total M has $mn + m + m(n-1) = 2mn$ elements. Finally, note that the second condition for M to be a matching is satisfied because each of the elements from W, X and Y are used exactly once.

The entire transformation requires the creation of $3(2mn)$ elements among three dimensions. The restricting set R contains the $2mn$ elements that will form M (if possible), and in addition, the mn elements left over from the T_i's, the $2m$ elements left over from the C_j's, and the $[2m(n-1)mn] - [m(n-1)] = m(n-1)(2mn-1)$ elements left over from G. Writing out all of these elements can be done in polynomial time. Once R has been specified, we can submit it to a 3DM-oracle that will find a matching if and only if F is satisfiable, and this submission need only occur once. Thus the transformation takes place in polynomial time, completing the reduction. We conclude that 3DM is NP-complete.[12]

5 Conclusion

In this paper we've seen the basis of all modern thinking about algorithmic problems. Though no programmers today feel the need to translate their code into the language of a Turing machine (thanks in part to faith in Church's Thesis) it is still the baseline model that allows us to make claims about the time complexity of algorithms. In addition, more and more NP-complete problems are arising as the result of scientific research in many different fields, lending ever more urgency to the drive to confirm or disprove the suspicion that P $\neq$ NP.

NP-completeness and polynomial-time reductions are central concepts in complexity theory. Nonetheless, just as this paper has only scratched the surface of the thousands of known NP-complete problems, these two concepts are only the barest beginnings of the results we can obtain from complexity theory. For example, we mentioned that to find a problem NP-complete is to throw cold water on our hopes of finding an efficient algorithm to solve it, but instead of giving up entirely we can create algorithms that efficiently generate solutions, with some known chance of failure or error. For algorithms like these, a whole new species of complexity classes emerges. Further, our results need not only be applicable to decision problems. It can be shown in many cases that the optimization and evaluation forms of a problem (for example, questions like "what is the largest clique in this graph?" and "what is the size of the largest clique in this graph?") are polynomial-time reducible to the decision form that we have examined here. A final example is the exploration of space complexity, which complements the time complexity results that we've seen and is of just as much importance to computer science, again despite the rapidly growing storage and memory capacity of modern digital computers. Complexity theory, in short, remains a rich field with many open problems and wide-ranging implications for math and science.

[12]Garey & Johnson pp. 50-3

References

Arora, Sanjeev and Boaz Barak. Computational Complexity.
Cambridge: Cambridge University Press, 2009.

Garey, Michael R. and David S. Johnson. Computers and Intractability.
New York: Bell Telephone Labs, Inc. 1979.

Homer, Steven and Alan L. Selman. Computability and Complexity Theory.
New York: Springer-Verlag, 2001.

Martin, John. Introduction to Languages and the Theory of Computation.
New York: McGraw-Hill, 2003.

Wegener, Ingo. Complexity Theory.
Berlin: Springer-Verlag, 2005.