Complexity Theory and The 3-Satisfiability Problem

T. Ian Martiny

April 29, 2012

1 Introduction

In this paper we will discuss Complexity Theory, some of its finer points and a specific problem know as the 3-Satisfiability (3SAT) problem. To understand some of the analysis behind the 3SAT problem we must first have some background knowledge on problems in general and how difficult certain problems can be. For this we begin with an introduction to Complexity Theory and some general types of problem difficulties. After this we will discuss the general Satisfiability (SAT) problem and then the 3SAT problem. After an introduction to these problems we will provide some analysis of the SAT and 3SAT problems and see how they relate to other problems.

2 Complexity Theory

Mathematics and Computer Science are two fields of study that mainly focus on solving problems, both applied and abstract. Anyone who has done work in either of those fields can attest that some problems are harder than others. And since these fields focus strongly on problems there must be some way to distinguish a problem as being 'difficult' or 'easy'. The issue with the classification is that at first it may seem to be subjective, that is in elementary school we may have thought multiplication problems were difficult but now we do not worry about those as much; this is the main focus of Complexity Theory. Complexity Theory deals with decision problems. A decision problem is one with given input that returns a 'yes' or 'no' answer and the answer can be verified. An example of a decision problem would be 'given two integers x and y does x divide y?' this is a decision problem because the answer is either 'yes' or 'no' and we have some method for determining if a given answer is correct. Complexity Theory classifies problems (and their solutions) based on how long the best known solution takes to report an answer, based on the input. There are multiple classes of problems but we will deal mainly with one, NP problems. To introduce these we must first begin with P problems. P stands for polynomial, P problems are ones for which there exists at least one algorithm to solve the problem, such that the number of steps of the algorithm is bounded by some polynomial in n where n is the input length. At first this definition seems to be convoluted, but it helps to examine some known P problems. Calculation of the greatest

common divisor of two integers is in P, and so is determining whether a given integer is prime. So what this means is that given an n digit number it takes some polynomial amount of time, in terms of n, to determine whether the number is prime or not. This could be any polynomial of n, say n^2-time or $n^{78234} - 7n^{27} + 2$-time, because the algorithm for determining whether a number is prime can be computed in some polynomial of n-time this problem is classified as being in P.

A counter part to P is NP, which are considered to be the more difficult problems. NP stands for non-deterministic polynomial, but can be defined as the set of problems that when given a solution, the solution can be verified as a solution in polynomial time. Again this definition seems strained and difficult, but is not too bad once it is worked out [2]. For example, say there is an instance of a problem and someone has generated a solution. If it can be verified as a solution (i.e. if it can be checked to be correct) in polynomial time (of n the input) then this is an NP problem. At first it seems that this is the same as P problems, which sort of makes sense, because all that is being discussed is more polynomial time solutions. And in fact it is easy to see that $P \subseteq NP$ because if our problem is P and we are given a potential solution we could easily forget the given solution and compute the solution with our polynomial time algorithm and check to see if the solutions are the same. But there are other problems that fall into NP as well, for example, suppose there is some problem that currently has no known polynomial time algorithm but we can easily verify the solution. Then at best we can classify this problem as NP, until a polynomial time algorithm is found (if it exists). An example of a current NP problem is the integer factorization problem: given an integer N and and integer M with $1 \leq M \leq N$ does N have a factor d with $1 < d < M$? Currently there is no polynomial time algorithm to determine if such a d exists, but it would be trivial given a d and N and M to determine if $d < M$ and if $d|N$.

A big current open problem in mathematics and theoretical computer science is to determine if $P = NP$, that is if the set of problems with polynomial time solutions is the same as the problems such that when given a solution we can check the validity of the solution in polynomial time. We have already shown the easy part of $P \subseteq NP$ but the converse is difficult to get a hold on. The significance of proving that $P = NP$ is that, if it were true then all of the problems that we currently have slow (non-polynomial time) solutions for have a more efficient solution, we just have to find it. Vice versa, if $P \neq NP$ then

there are some problems which cannot have efficient polynomial time solutions.

One more class of problems that we will be concerned with is a subset of NP, NP-complete. A problem, Q, is NP-complete if any other NP problem, say R, can be transformed into Q in polynomial time. Basically what this is saying is that if we can solve Q quickly then we can solve R quickly. The problems in NP-complete are currently unsolvable in polynomial time. Here we do not mean that we cannot solve any instance of this problem quickly. What we mean is that there is no algorithm that can solve every instance of the problem and complete in polynomial time. If any of the problems in NP-complete could be solved in polynomial time then we would be able to say that $P = NP$. A current NP-complete problem is solving quadratic recurrences, that is given a modulus n and a q if there exists an x such that $x^2 \equiv q \pmod{n}$. Because of the difficulty of solving this problem in general the quadratic residue problem is used in public-key cryptography to share sensitive information with some but hide it from others. Another NP-complete problem is the 3-SAT problem, which we will focus on in later sections, and prove is NP-complete [1].

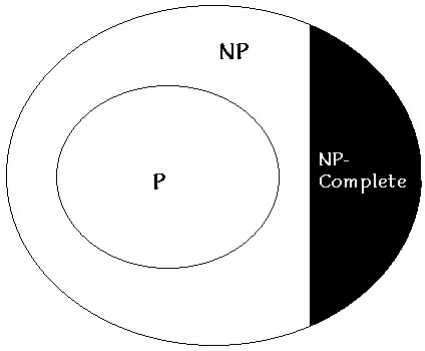

Figure 1: Our Universe of Problems

Figure 1 shows a possible diagram of how the space of problems looks. It is only a possible diagram due to the current problem of $P = NP$ still being open, it could be that all our problems could be solved in polynomial time. As a refresher, when we are dealing with problems and their solutions, we consider some problem, Q. If there is at least one algorithm that can solve all instances of Q, with the worst case taking polynomial time (based on the size of the input) then we say that Q (and its solution) is

a polynomial time or P problem. If there is currently no algorithm whose worst case can solve every instance of Q in polynomial time, but given a potential solution to Q we can check the solution as correct in polynomial time we say Q is a non-deterministic polynomial or NP problem. Furthermore, if in addition to being an NP problem we can turn every other NP problem into an instance of Q then we call Q an NP-complete problem, we will see some examples of this coming up.

3 Satisfiability Problems

In order to understand any of the Satisfiability problems we must first define a few terms. A clause is a finite collection of boolean (true or false) variables or their complements. A clause is satisfiable if and only if there is some assignment of truth values to the variables to make the whole clause true. The Satisfiability problem involves boolean expressions written using only boolean variables, parenthesis, logical *and*, logical *or*, and logical *not*. For example if x_1 and x_2 are two boolean variables we generally use the symbol $\wedge$ to represent logical *and*, and we use $\vee$ to represent logical *or*, and we use $\neg$ to represent logical *not*. Thus $\neg x_1 \vee x_2$ would represent '*not* x sub one *or* x sub 2'. The Satisfiability problem is as follows: Given a finite set of k clauses, determine whether there is an assignment of truth values to the variables appearing in the clauses which makes all of the clause true. A particular instance of the SAT problem is

$$\{A \vee B, A \vee D, \neg(B \wedge C \wedge E), B \vee E, \neg(A \wedge E)\}$$

where A, B, C, D, E represent boolean variables. A solution to this SAT problem is A =True, B = True, C = False, D = True, E = False. With this assignment of truth values we have that each clause is true and thus satisfied. Given that once a solution to the above problem was supplied it was easy to check that the solution was valid,the SAT problem is certainly in NP. In 1971 Stephen Cook published his paper 'The Complexity of Theorem Proving Procedures' in which he proved that the SAT problem is NP-complete. The idea of the proof is simple; in order to show that the SAT problem is NP-complete we must first show the SAT problem is in NP (which we have already done). Next we must take an arbitrary problem in NP and show that we can convert the problem into an instance of the SAT problem;

afterwards we must show that if we have a solution to this instance of the SAT problem we also have a solution to the original NP problem. A more concise way to state this is that we must convert the NP problem into an instance of the SAT problem without any loss of information.

We go into more depth on problem conversion here. A decision problem D_1 can be converted into another decision problem D_2 if there is an algorithm which takes input of an instance of D_1 and returns an instance of D_2, such that the instance of D_2 is satisfied (or positive) if and only if the instance of D_1 one is positive [3]. Thus if we have an algorithm to solve D_2 then we also have one to solve D_1; we can first convert an instance of D_1 into an instance of D_2 then apply our algorithm to solve D_2, if it returns a positive for D_2 then we know that we have a positive for D_1. Remember here we are dealing with decision problems which only return a positive or negative (think true or false) result. Essentially decision problems ask yes or no questions. Because we are able to convert instance of D_1 into instances of D_2 we can say that the complexity of D_1 is at most the complexity of D_2 plus the complexity of our conversion algorithm. Thus if we have an algorithm to solve D_2 in polynomial time and we have a polynomial algorithm to convert instances of D_1 into instances of D_2 then we essentially have a solution algorithm to D_1 in polynomial time, because any two polynomials added together result in another polynomial [4].

This analysis of problem conversion along with Cook's Theorem gives us that if we could solve the SAT problem in polynomial time then we could solve any problem in NP in polynomial time. Furthermore if we can solve any problem in NP then we have that $P = NP$, so we have effectively reduced the monumental task of solving every problem in NP in polynomial time (in order to show that $P = NP$) to solving just the SAT problem in polynomial time. Surely this should be a much easier task? However despite all the work that has been done on the SAT problem there is no general algorithm (yet) which can solve the general SAT problem in polynomial time. That being said there are algorithms that can solve a large enough subset of SAT problems to be helpful in certain areas, such as automatic theorem proving.

Perhaps the reason the SAT problem has yet to be solved in general is that there are too many variables per clause. In the SAT problem we can have any number of clauses and any number of variables in each clause, maybe if we had less variables per clause we would be able to solve this problem. We next

consider the 3SAT problem which handles this very case. Before we move on, we pause first to remember what we gained from this section. The Satisfiability or SAT problem is a decision problem which asks if given a series of boolean variables grouped together into clauses, where within each clause the variables can be meshed together by logical *and* or logical *or*, and each clauses is meshed together by logical *and*, is there an assignment of boolean values to each variable such that the whole statement is true? Cook's Theorem showed that not only is the SAT problem certainly in NP (we can verify solutions quickly) but it is NP-complete. That is given any instance of any NP problem we can convert it into an instance of the SAT problem and the result of the instance of the SAT problem is the result of the original problem. The significance of this is that now if we were trying to show $P = NP$ (or $P \neq NP$) then we would only need to find an algorithm which solves the general SAT problem in polynomial time (or show one does not exist) which reduces the amount of work from showing this for all NP problems to just one. Even with this reduction the SAT problem still has no general solution. Also gained from this section is the concept of problem conversion. We can convert instances of one decision problem to another where the result of the latter is true if and only if the result of the initial is true. This concept was instrumental to the proof of Cook's Theorem (not covered here), and will be instrumental later in this paper as well. We now examine a subset of the SAT problem known as the 3SAT problem.

4 3-Satisfiability Problem

We now restrict the SAT problem to the 3-Satisfiability (3SAT) problem to see if our restrictions will make a solution appear. We must be careful when we restrict the SAT problem to not lose the NP-Completeness of the problem. We define the 3SAT problem as a list of k clauses which are grouped together with logical *and*. So far this is the same as the SAT problem, however in 3SAT we require that each clause only contain 3 variables, the variables need not be distinct, and also the same as SAT we can negate variables. This, for example, is an instance of a 3SAT problem

$$\{(x_1 \vee x_2 \vee \neg x_3), (x_2 \vee x_3 \vee \neg x_4), (x_1 \vee \neg x_2 \vee x_4)\}$$

with one solution being $x_1 = \text{True}$, $x_2 = \text{True}$ (which make all three clauses true) so we have no restriction on x_3 or x_4. However note that our example above:

$$\{A \vee B, A \vee D, \neg(B \wedge C \wedge E), B \vee E, \neg(A \wedge E)\}$$

is not a 3SAT problem as there are clauses with only two variables in them. Since 3SAT has more restrictions it should be clear that every 3SAT problem is also a valid SAT problem, but can we say the same of the converse? To do this would be analogous to showing that the 3SAT problem is NP-Complete, since if SAT is NP-Complete we can rewrite any NP problem as an instance of the SAT problem in polynomial time and we will show we can rewrite any instance of the SAT problem as an instance of the 3SAT problem in polynomial time. Thus we will have effectively shown that any NP problem can be rewritten as an instance of the 3SAT problem in polynomial time, making it NP-Complete.

To show that 3SAT is in NP we simply note that given a possible solution to a 3SAT problem we need only plug in the value of the variables and check the logical conjunctions, which is easily done in polynomial time. Now we show a polynomial time reduction of the SAT problem to the 3SAT problem. Let the given instance of SAT contain the collection $C = \{c_1, c_2, c_3, \ldots, c_m\}$ of m clauses over the n variables: $u_1, u_2, \ldots, u_n$. We construct a collection C' of 3 literal clauses of a set of variable that contains all the original variables plus sets of additional variables.

In our general SAT problem we were not concerned with how the variables were grouped together, they could be grouped with logical *and* or logical *or*. To make our conversion to the 3SAT problem simpler we are going to assume that in our instance of the SAT problem the variables are grouped together with only logical *or*. Even though this may seem as though we are only treating a specific case of the SAT problem we actually have no loss of generality. For example, given a statement $x_1 \wedge x_2$ (x_1 and x_2) we can rewrite this as a set of clauses $\{(x_1 \vee x_2), (\neg x_1 \vee x_2), (x_1 \vee \neg x_2)\}$. Note that this set is only true when $x_1 \wedge x_2$ is true. Even though we are requiring that there be no logical *and* within clauses we still use logical *and* to join separate clauses. The same idea holds with more than two variables (and negation of variables).

To determine which additional variables need to be added we must look at cases. We examine the

case where a clause from the SAT problem contains only one variable, two variables, three variables and then the general case of any number more than three variables. Now let $c_i \in C$ and k represent the number of variables in a clause thus:

If $k = 1$ then $c_i = \{z_1\}$. We create two additional variables $\{y_{i,1}, y_{i,2}\}$ and form the collection $C_i' = \{\{z_1, y_{i,1}, y_{i,2}\}, \{z_1, y_{i,1}, \neg y_{i,2}\}, \{z_1, \neg y_{i,1}, y_{i,2}\}, \{z_1, \neg y_{i,1}, \neg y_{i,2}\}\}$. In this case we have added two variable and a number of clauses. Thus from one clause in the SAT problem, we have four clauses in the 3SAT problem. Note however that we have not given an assignment to either of the y variables, which we will handle later.

If $k = 2$ then $c_i = \{z_1, z_2\}$. We use one additional variable $\{y_{i,1}\}$ and form the collection $C_i' = \{\{z_1, z_2, y_{i,1}\}, \{z_1, z_2, \neg y_{i,1}\}\}$. In this case we add only one variable and have two clauses, but again at least one of the clauses is true.

If $k = 3$ then $c_i = C_i' = \{z_1, z_2, z_3\}$ since we already have three variables in our clause we have no need to make any changes or modifications.

If $k > 3$ then we use additional variables $\{y_{i,1}, y_{i,2}, \ldots, y_{i,k-3}\}$. Here we form the collection

$$\{\{z_1, z_2, y_{i,1}\}, \{\neg y_{i,1}, z_3, y_{i,2}\}, \{\neg y_{i,2}, z_4, y_{i,3}\}, \ldots, \{\neg y_{i,k-3}, z_{k-1}, z_k\}\}$$

.

The work we have done so far is very nice but it means nothing if in doing so we have changed when the problem is true. So we cannot arbitrarily assign true and false values to our new y variables, we must be systematic about it; this assignment is however arbitrary when $k \leq 3$. Note that we are trying to show that whenever C is satisfiable then C' is satisfiable also. So we assume that we know some assignment in our original SAT problem to make the clauses satisfiable and we use this to decide what values the y variables should take on. Thus if we have a clause with one, two, or three variables that is satisfiable in C then no matter what value our new y variables take on, the same assignment to the original variables is satisfiable. So we only concern ourselves with the case that $k > 3$. Here again we consider cases based on how we grouped our new variables.

If in the original SAT problem z_1 or z_2 are true then assign false to all additional y variables. In this

case the first clause which has $\{z_1, z_2, y_{i,1}\}$ is true and every subsequent clause contains a negation of a y variable so each clause then has a true variable.

If z_{k-1} or z_k is true then assign true to all additional y variables. In this case the last clause which has $\{\neg y_{i,k-3}, z_{k-1}, z_k\}$ is true and every previous clause contains a y variable with no negation so it has at least one true variable.

Otherwise, if z_l is true, assign $y_{i,j}$ the value true when $1 \leq j \leq l-2$ and false when $l-1 \leq j \leq k-3$. In this case the third literal of each clause preceding the clause that contains z_l is true while the first literal in every clause after the one containing z_l is true.

By our above rules all of the clauses in C' are satisfiable. Conversely, if all the clauses in C' are satisfied by some truth assignment to the variables (which include our original variables) then certainly C is satisfied. So we have that C' is satisfied if and only if C is satisfied. We pause again to look at our previous two examples and to assign truth values to each of our new variables.

We pause at this instance of our proof to consider some examples of SAT problems and use our rules to turn them into instances of 3SAT problems to help with what our additional variables are doing. Note here that we have been using z variable to represent the variables in a clause. We have begun each clause with z_1 but in reality each clause could begin with any variable (as we will show in the examples), thus a convenient way to think about the z variable is as a numbering variable so z_1 would be 'the first variable in the clause', likewise z_2 would be 'the second variable in the clause', etc.

Example 1: Let $c_1 = \{\neg u_1, u_2, u_3\}$, $c_2 = \{\neg u_2, u_3\}$, $c_3 = \{u_1, \neg u_2, u_3, \neg u_4\}$ over the variables $\{u_1, u_2, u_3, u_4\}$. Remember we have made the modification to the SAT problem where all variables within a clause are grouped together with logical *or*, so the $\vee$ symbol is unneeded now. This instance has the solution of $u_1 =$ false, $u_2 =$ false and the values for u_3 and u_4 being unimportant. When we apply our rules to convert this to a 3SAT problem we arrive with $C'_1 = \{\neg u_1, u_2, u_3\}$ which is unchanged because c_1 already has three variables. $C'_2 = \{\{\neg u_2, u_3, y_{2,1}\}, \{\neg u_2, u_3, \neg y_{2,1}\}\}$ here we have added $y_{2,1}$ and put it with the original set of variables and put its negation with a separate set of the original variables. Finally, $C'_3 = \{\{u_1, \neg u_2, y_{3,1}\}, \{\neg y_{3,1}, u_3, \neg u_4\}\}$. Thus we went from our original SAT problem with four

variables and three clauses to a 3SAT problem:

$$\{\{\neg u_1, u_2, u_3\}, \{\neg u_2, u_3, y_{2,1}\}, \{\neg u_2, u_3, \neg y_{2,1}\}, \{u_1, \neg u_2, y_{3,1}\}, \{\neg y_{3,1}, u_3, \neg u_4\}\}$$

with six variables and five clauses. At this point we consider the y variables as dummy variables; we have no idea how they evaluate (true or false) but can only follow the given rules to rewrite SAT instances to 3SAT instances as given above. We provide rules for assigning values to the y variables later on.

Example 2: Let $c_1 = \{x_1, x_2\}$, $c_2 = \{x_3\}$, $c_3 = \{x_3, x_4, \neg x_2\}$, $c_4 = \{\neg x_3, \neg x_1, x_4, x_2, x_5\}$, $c_5 = \{x_5, \neg x_3, x_2, \neg x_4\}$ over the variables x_1, x_2, x_3, x_4, x_5. This has a solution of true when $x_1 =$ false, $x_2 =$ true, $x_3 =$ true $x_4 =$ false and $x_5 =$ true. After applying our rules to the collection of clauses we get that $C_1' = \{\{x_1, x_2, y_{1,1}\}, \{x_1, x_2, \neg y_{1,1}\}\}$, $C_2' = \{\{x_3, y_{2,1}, y_{2,2}\}, \{x_3, y_{2,1}, \neg y_{2,2}\}, \{x_3, \neg y_{2,1}, y_{2,2}\},$
$\{x_3, \neg y_{2,1}, \neg y_{2,2}\}\}$, $C_3' = \{x_3, x_4, \neg x_2\}$, $C_4' = \{\{\neg x_3, \neg x_1, y_{4,1}\}, \{\neg y_{4,1}, x_4, y_{4,2}\}, \{\neg y_{4,2}, x_2, x_5\}\}$, $C_5' = \{\{x_5, \neg x_3, y_{5,1}\},$
$\{\neg y_{5,1}, x_2, \neg x_4\}\}$. Thus we had our original SAT problem with five variables and five clauses to a 3SAT problem:

$$\{\{x_1, x_2, y_{1,1}\}, \{x_1, x_2, \neg y_{1,1}\}, \{x_3, y_{2,1}, y_{2,2}\}, \{x_3, y_{2,1}, \neg y_{2,2}\}, \{x_3, \neg y_{2,1}, y_{2,2}\}, \{x_3, \neg y_{2,1}, \neg y_{2,2}\},$$
$$\{x_3, x_4, \neg x_2\}, \{\neg x_3, x_1, y_{4,1}\}, \{\neg y_{4,1}, x_4, y_{4,2}\}, \{\neg y_{4,2}, x_2, x_5\}, \{x_5, \neg x_3, y_{5,1}\}, \{\neg y_{5,1}, x_2, \neg x_4\}\}$$

with eleven variables and twelve clauses. We now continue on with our conversion algorithm by defining true or false values to the newly created y variables.

Now that we have converted some instances of the SAT problem into instances of the 3SAT problem, we revist each example and assign truth-values to each variable to satisfy the statements.

Example 1 (cont.): We began with $c_1 = \{\neg u_1, u_2, u_3\}$, $c_2 = \{\neg u_2, u_3\}$, $c_3 = \{u_1, \neg u_2, u_3, \neg u_4\}$ over the variables $\{u_1, u_2, u_3, u_4\}$. Which was satisfied when $u_1 =$ false, $u_2 =$ false and the values for u_3 and

u_4 being unimportant, so lets say $u_3 =$ false and $u_4 =$ false. We ended with the problem

$$\{\{\neg u_1, u_2, u_3\}, \{\neg u_2, u_3, y_{2,1}\}, \{\neg u_2, u_3, \neg y_{2,1}\}, \{u_1, \neg u_2, y_{3,1}\}, \{\neg y_{3,1}, u_3, \neg u_4\}\}$$

and now we assign values to $y_{2,1}$, and $y_{3,1}$ based on our given rules. Our only clause which contained more than three variables is the last one, so we need to only consider what $y_{3,1}$ should be. Because the second clause (in the original SAT) problem only contained two variables and was satisfied we can just arbitrarily choose false for $y_{2,1}$. In the third clause of our original problem z_2 corresponds to $\neg u_2$, and because we found that $u_2 =$ false satisfied our original problem this means that $z_2 =$ true ($\neg$ false). Because z_2 is true we have that all y variables can be assigned false and have every clause be satisfied. And indeed with the assignments $u_1 =$ false, $u_2 =$ false, $u_3 =$ false, $u_4 =$ false, $y_{2,1} =$ false, and $y_{2,2} =$ false satisfies our 3SAT problem.

Example 2 (cont.): We began with $c_1 = \{x_1, x_2\}$, $c_2 = \{x_3\}$, $c_3 = \{x_3, x_4, \neg x_2\}$, $c_4 = \{\neg x_3, \neg x_1, x_4, x_2, x_5\}$, $c_5 = \{x_5, \neg x_3, x_2, \neg x_4\}$ over the variables x_1, x_2, x_3, x_4, x_5. Which was satisfied when $x_1 =$ false, $x_2 =$ true, $x_3 =$ true $x_4 =$ false and $x_5 =$ true. We ended with the problem:

$$\{\{x_1, x_2, y_{1,1}\}, \{x_1, x_2, \neg y_{1,1}\}, \{x_3, y_{2,1}, y_{2,2}\}, \{x_3, y_{2,1}, \neg y_{2,2}\}, \{x_3, \neg y_{2,1}, y_{2,2}\}, \{x_3, \neg y_{2,1}, \neg y_{2,2}\},$$
$$\{x_3, x_4, \neg x_2\}, \{\neg x_3, \neg x_1, y_{4,1}\}, \{\neg y_{4,1}, x_4, y_{4,2}\}, \{\neg y_{4,2}, x_2, x_5\}, \{x_5, \neg x_3, y_{5,1}\}, \{\neg y_{5,1}, x_2, \neg x_4\}\}$$

we now assign truth values to our new variables. Because our first three clauses in the original problem contain less than four variables and were satisfied, we can simply assign false to $y_{1,1}$, $y_{2,1}$, and $y_{2,2}$ because they are unimportant (clauses which contain those variables are already satisfied). Thus we first consider our fourth and fifth clauses from the original problem and variables $y_{4,1}$, $y_{4,2}$ and $y_{5,1}$. Because $z_2 = \neg x_1$ and $x_1 =$ false so $z_2 =$ true in c_4 we can assign $y_{4,1}$, and $y_{4,2}$ the value of false and have all relevant clauses be true. Similarly in c_5 because $z_1 = x_5$ and x_5 is assigned true then z_1 is assigned true so $y_{5,1}$ can be assigned false and every relevant clause evaluates to true.

We certainly have a reduction of a SAT problem to a 3SAT problem, but remember the importance

of NP-Complete: the reduction must be polynomial in time. Remember in our original (general, i.e., not one of the examples) SAT problem that we had $C = \{c_1, c_2, c_3, \ldots, c_m\}$ over n variables the number of 3 literal clauses in C' is bounded above by a polynomial in nm, so we have this procedure taking some polynomial amount of time as desired. By the above proof we see that the 3SAT problem is indeed NP-Complete [5].

In this section we introduced the 3SAT problem as a subset of the SAT problem. That is the 3SAT problem is a collection of k boolean clauses where each clause contains only 3 literals (boolean variables) which are disjuncted by logical *or*. Each clause is conjuncted by logical *and*; our question is, 'is there an assignment of truth values to the boolean variables such that each clause has a truth value of true'? In addition to introducing the 3SAT problem we also introduced a new way to prove that problem is NP-Complete. Before (as in proving SAT was NP-Complete) when proving a problem Q is NP-Complete, we had to take an arbitrary NP problem and prove that we could reduce it to an instance of the problem Q in polynomial time and such that we got a result of 'yes' in the original NP problem if and only if we got a result of 'yes' in the problem Q. Now that we can use Cook's Theorem which shows that SAT is NP-Complete we need only show that we can reduce any instance of the SAT problem to our general NP problem in polynomial time with a result of 'yes' in the SAT problem if and only if we would get a result of 'yes' in the original problem. This may not seem like much but it is a great deal because before we had no information on the original NP problem to start with but we know at least some things about the SAT problem to work with. The reason we do not need to concern ourselves with general NP problems anymore is that we know we can reduce any NP problem to an instance of the SAT problem in polynomial time, so we can reduce the SAT problem to an instance of our new problem in polynomial time the amount of time to go from any general NP problem to our new problem would be the time to go from the NP problem to SAT (polynomial) plus the time to go from SAT to the new problem (polynomial). And of course adding two polynomials results in a new polynomial. With this method of proof we were able to show that even the 3SAT problem is NP-Complete by reducing the SAT problem to an instance of the 3SAT problem in polynomial time. Now that we have this reduction if we were to show some other problem were NP-Complete all we would need to do is either reduce an instance of

3SAT or SAT to the new problem in polynomial time (preserving our 'yes' answers of course). So in order to prove (or disprove) that $P = NP$ we need only prove (or disprove) that there exists an algorithm to solve the general 3SAT problem in polynomial time. Next we talk of some of the advances in this last step.

5 Conclusion

Unfortunately even though the 3SAT problem seems simpler there is still no known polynomial time algorithm to solve the general 3SAT problem. In fact, this should not be a surprise since we showed that the 3SAT problem is NP-Complete and have mentioned before that the question of if $P = NP$ is still open.

What we have learned here is that there are classes of problem that we group by difficulty. The simpler problems we call P as long as we can solve all instances of the problem in polynomial time, based on the size of the input. An example of a P problem is determining if a given integer is prime. A more interesting section of problems is the set NP. NP problems are ones in which if we are given a solution to a problem we can verify the solution is correct in polynomial time (again based on the size of the input). It was simple to show that $P \subseteq NP$ and we commented on how whether $P = NP$ or not is still an open problem, though the general consensus is that it is not. When we say that there does not exist a polynomial time algorithm for a problem we do not mean that we can not solve any instances of the problem; in fact this is generally not the case, with the SAT problem and Integer Factorization problem we can solve these in polynomial time for moderately sized input, but there does not exist an algorithm which can solve every instance of the problem (regardless of size) in polynomial time. An example of an NP problem was integer factorization. We also examined a third subset of problems known as NP-Complete; which is the set of problems where any NP problem can be reduced to an NP-Complete problem such that a positive result of the NP-Complete problem occurs only when there would be a positive result to the original NP problem. One example of an NP-Complete problem is the SAT problem.

After describing the types of problems we introduced the Satisfiability problem. This problem deals with clauses of literals (boolean variables), the clauses are grouped together by logical *and*, our goal is to

find if there exists some assignment of truth values to each variable such that each clause (and thus the whole statement) evaluates to true. After looking at some examples of this problem we quoted Stephen Cook's Theorem which says that the SAT problem is NP-Complete. The method of proof he used was to first show that SAT was in NP, which means we can easily verify solutions. Then he had to take any general problem Q from NP and reduce it to an instance of SAT in polynomial time. This is a difficult task to accomplish and his proof is quite lengthy. However because he has proved it we gain more than just the knowledge of SAT being NP-Complete. We gain an easier way to classify problems as NP-Complete. We no longer have to show any general NP problem can be converted, we only have to convert the SAT problem in polynomial time because we know we can get any NP problem to the SAT problem in polynomial time.

After we examined the SAT problem we moved on to the 3SAT problem which is the same as the SAT problem except that within each clause we can only have 3 literals, where as in SAT the number could be any finite number. Using the method discussed we proved that the 3SAT problem is indeed NP-Complete. At first this seems like it should be an easier problem to solve but unfortunately when we reduce the number of variables in each clause we also increase the number of clauses.

References

[1] Michael Gary and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[2] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. Springer Science+Business Media, LLC, 2010.

[3] Roberto Sebastiani. NP-Completeness and Cook's Theorem, 2002.

[4] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996.

[5] M. Warmuth. NP-Completeness: Some Reductions, 2010.