A Polynomial Time Algorithm for 3SAT

Robert Quigley

October 3, 2024

Abstract

This paper presents a polynomial time algorithm for 3SAT using the idea of Implication which allows for the derivation of contradicting 1-terminal clauses iff the instance is unsatisfiable. The idea is that any two clauses which contain the same terminal which is positive in one clause and negated in the other can imply a new clause which consists of all the terms in either clause except for the opposite form terms. Two 1-terminal clauses which contain opposite form terms are called contradicting 1-terminal clauses and their existence implies the instance is unsatisfiable.

Introduction

This paper presents and explains a polynomial time algorithm for 3SAT. In order to convey the ideas in their entirety, it first presents some concepts and definitions to familiarize the reader with an appropriate way of thinking. It then continues to explore helpful ideas about 3SAT before finally presenting and explaining the algorithm. It presents definitions as they naturally arise as to not front load the reader with an excessive amount of information without any context. This paper is written for those with an understanding of 3SAT or at least variables, logic, algorithms, and time complexities.

Importance and Implications

In short, Karp points out that the existence of a polynomial time algorithm for 3SAT implies P = NP. In [1], Karp defines the class P, the class NP, and

the problem 3SAT. The class P refers to "the class of languages recognizable in polynomial time by one-tape deterministic Turing machines." The class NP refers to "the class of languages recognizable in polynomial time by one-tape nondeterministic Turing machines."

In other words, P refers to the class of problems whose algorithms have time complexities which are polynomial with respect to the length of the input. NP refers to the class of problems whose solutions are polynomial with respect to the length of the input. However, the search for the solution (done by the algorithm) could take up to exponential time. This is the question of P vs NP: for each problem in NP, does there exist an algorithm which produces a solution in polynomial time?

Regarding 3SAT, Karp introduces the SATISFIABILITY problem and shows how

P = NP $\iff$ SATISFIABILITY $\in$ P.

Karp goes on to show other problems that can play the role of SATISFIABILITY in the two way implication above. He calls them "complete". These problems are now more commonly referred to as "NP-complete". The problem 3SAT is NP-complete. Karp defines 3SAT as follows:

> SATISFIABILITY WITH AT MOST 3 LITERALS PER CLAUSE
>
> INPUT: Clauses $D_1, D_2, ..., D_r$, each consisting of at most 3 literals from the set $\{u_1, u_2, ..., u_m\} \cup \{\bar{u_1}, \bar{u_2}, ..., \bar{u_m}\}$
>
> PROPERTY: The set $\{D_1, D_2, ..., D_r\}$ is satisfiable.

In an earlier definition of SATISFIABILITY, Karp uses the following property to define a set of clauses being satisfiable:

> PROPERTY: The conjunction of the given clauses is satisfiable; i.e., there is a set $S \subset [\{u_1, u_2, ..., u_n, \bar{u_1}, \bar{u_2}, ..., \bar{u_m}\}]$ such that
>
> a) S does not contain a complementary pair of literals and
>
> b) $S \cap [D_k] \neq \phi, k = 1, 2, ..., [r]$

This is to say that there is a set, S, which is a subset of literals and their complements in which S does not contain both a literal and its complement and every clause contains at least one literal which exists in S.

It has been helpful to think of this problem slightly differently. When constructing the set, S, if S contains a literal from the set $\{u_1, u_2, ..., u_m\}$,

we can think of that literal being assigned the value of True. And if S contains a literal from the set $\{\bar{u_1}, \bar{u_2}, ..., \bar{u_m}\}$, we can think of that literal being assigned the value of False.

Thinking of literals as boolean variables (called terminals), we can think of a clause as containing three terminals combined by logical OR operators. Similarly, we can think of an instance of 3SAT as a set of clauses combined by logical AND operators. The question of the existence of S is now the question of the existence of a value (True or False) for each terminal such that the instance evaluates to True.

Instead of a strict translation from the word "literals", this paper uses the following vocabulary to refer to a literal depending on context: The literals in the set $\{u_1, u_2, ..., u_m\}$ will be referred to as *terminals* when discussing the number of terminals in an instance or when assigning them a value (True or False). When discussing literals existing in a clause before a concrete value is known, we refer to literals in the set $\{u_1, u_2, ..., u_m\}$ as *positive terms* and the literals in the set $\{\bar{u_1}, \bar{u_2}, ..., \bar{u_m}\}$ as *negative terms*.

The problem of 3SAT is to determine whether the property holds true for any given instance. As stated above, a polynomial time algorithm for 3SAT implies P = NP.

Reiteration and Terminology

This section aims to reiterate the problem of 3SAT as well as introduce some terminology and notation used in this paper.

An *instance* of 3SAT consists of a set of *clauses* combined by logical AND operators.

A *clause* is a set of exactly three unique *terminals* combined by logical OR operators. Each terminal may also have the logical NOT operator applied to it.

A *terminal* is a variable whose value could be assigned to True or False.

When discussing terminals existing in a clause before a concrete value is known, we refer to terminals as *positive terms* and their complements as *negative terms*. We may also refer to a clause as containing a certain number of *terms*, considering both positive and negative terms.

An *assignment* is a set of values assigned to each terminal.

An instance is said to be *satisfiable* if there exists an assignment such that the instance evaluates to True. We call this the *satisfying assignment*.

An instance is said to be *unsatisfiable* if there does not exist an assignment which allows the instance to evaluate to True.

The notation used to represent clauses in the paper is as follows:

1. clauses are surrounded by a single pair of square brackets
2. terms are written as letters, separated by commas
3. negative terms are prepended with a minus sign
4. when clauses are named, they are generally named using a minimal number of capital letters

Some example clauses follow:

$A := [a, b, c]$

$B := [a, b, -c]$

where A is a clause which contains the terms a, b, and c and B is a clause which contains the terms a, b, and the complement of c.

Important Note: Often this paper presents example clauses and points out the positive or negative terms in those clauses. This is always done without loss of generality as it is not strictly referring to a positive or negative term in the absolute sense. The only important aspect about positive or negative terms is how they behave when considering the same terminal in two different forms. For example, a clause presented as

$A := [a, -b, c]$

could represent a clause with any three terms in either form. The positive or negative nature of a, b, and c are only important when considering another clause containing the same term and whether that term's form is the same or opposite.

An Exploration of Ideas

This section presents ideas whose combined value is critical for proving the polynomial time nature of the algorithm presented later in this paper.

Blocking an assignment

Claim: A clause, C, is said to *block* an assignment, A, if the presence of C in the instance prevents A from being the satisfying assignment.

Proof. Recall all clauses within an instance are combined by logical AND operators.

Therefore all clauses within an instance must evaluate to True for the instance to evaluate to True.

Recall all terms within a clause are combined by logical OR operators.

Therefore at least one term in a clause must be True in order for the clause to evaluate to True.

If all terms in a clause are False, the clause evaluates to False.

If any clause is False, the instance evaluates to False.

Any assignment which sets all the terms in a clause to False will therefore never be the satisfying assignment. We say this assignment is blocked by the clause whose terms are all set to False. □

The number of possible assignments

Claim: For a given instance with n terminals, there are 2^n possible assignments.

Proof. Recall each terminal has two options for its value: True or False.

Because there are two options for each terminal and there are n terminals in a given instance, there are 2^n possible assignments for that given instance. □

The number of assignments blocked per clause

Claim: In an assignment with n terminals, each clause of length k blocks 2^{n-k} assignments.

Disclaimer: In an instance of 3SAT, the given clauses all contain exactly three terms, but it is useful in later proofs to consider clauses with a fixed, yet arbitrary length, k.

Proof. Recall an assignment is blocked by a clause if that assignment assigns False to all the terms in that clause.

Considering assignments blocked by a clause of length k, all of these assignments fix the values for the k terminals in this clause.

The remaining $n - k$ terminals have two options: True or False.

Recall an assignment exists for all the ways to assign True or False to all the terminals in an instance.

Therefore there exists one blocked assignment for each possible combination of True or False assigned to the remaining $n - k$ terminals.

This results in 2^{n-k} assignments being blocked by a clause of length k. □

Assignments blocked by fixing a terminal's value

Claim: For any clause, C, if we select a terminal, t, that's not in C, then half of the assignments blocked by C will assign True to t and the other half will assign False to t.

Proof. Let the clause and term be defined as follows:

$C := [a, b, c, ...]$

in which C is a clause of length k, and t is a terminal which does not exist in C.

We want to show that half of the assignments blocked by C assign True to t and the other half assign False to t.

Let's fix the value of t to True.

Considering the assignments blocked by C which assign True to t, it is seen these assignments fix the value of $k+1$ terms and allow for the remaining $n-k-1$ terminals to be True or False.

This means there are 2^{n-k} assignments blocked by C and there are 2^{n-k-1} assignments blocked by C which assign True to t.

It is seen that half of the assignments blocked by C assign True to t.

Similarly, it can be seen that half of the assignments blocked by C assign False to t.

Since t cannot be assigned both True and False, these sets are disjoint.

Therefore, half of the assignments blocked by C assign True to t and the other half assign False to t. □

Reduction

Claim: Two clauses which are identical except for one term, i, which is positive in one clause and negative in the other, *imply* another clause which is composed of all of the terms in the original clauses except for i and $-i$.

Definition: A clause or set of clauses is said to *imply* another clause if the latter clause can be added to the instance without blocking any additional assignments.

Definition: We say the first two clauses *reduce to* the third clause. Sometimes the first two clauses are referred to as *inputs* of the implication and the third clause is referred to as the *output* of the implication.

Definition: The term which is positive in one clause and negative in the other is sometimes referred to as the *popped term* or as an *opposite form term*.

Proof. Let the clauses be defined as follows:

$A := [a, b, c, ..., i]$

$B := [a, b, c, ..., -i]$

$C := [a, b, c, ...]$

where A, B, and C share the identical set of terms, $a, b, c, ...$ and i is positive in A and negative in B.

We want to show that A and B imply C.

Recall that if we take a terminal that's not in a clause, then half of the assignments blocked by that clause assign True to that terminal and the other half assign False to that terminal.

Consider the clause, C, and the terminal, i, which is not in C.

Half of the assignments blocked by C assign True to i and the other half assign False to i.

The assignments blocked by C which assign False to i are blocked by A.

The assignments blocked by C which assign True to i are blocked by B.

Notice A and B block both halves of the assignments blocked by C.

In other words, A and B block all of the assignments blocked by C and it can be said that A and B imply C. □

Expansion

Claim: A clause, C, expands to a clause, D, if all the terms in C exist in D and there is at least one term in D which is not in C.

Definition: We say C *expands to* D. It may be said that C implies D.

Definition: Any terms in D which are not in C are called *expanded terms.*

Proof. Let the clauses be defined as follows:

$C := [a, b, c, ...]$

$D := [a, b, c, ..., d, e, f, ...]$

where $a, b, c, ...$ exist in both C and D and $d, e, f, ...$ exist only in D.

We want to show that C implies D.

Recall a clause blocks assignments which assign False to all terms in that clause.

Recall also that the blocked assignments assign every possible combination of True and False to every terminal that is not in the clause.

Consider the assignments blocked by C.

C blocks assignments where $a, b, c, ...$ are all False.

Additionally, there exists a blocked assignment for each possible combination of True and False for every terminal that is not in C.

Specifically, there exists assignments which set $a, b, c, ..., d, e, f, ...$ to False as well as every possible combination of True or False for the remaining terms.

Notice these are exactly the assignments blocked by D.

Therefore all of the assignments blocked by D are blocked by C.

It can be said that C implies D. □

Implication

Claim: Two clauses which contain the same term which is positive in one clause and negative in the other imply a third clause which consists of all the terms in the first two clauses except for the opposite form terms.

Proof. Let the clauses be defined as follows:

$A := [a, b, c, ..., i]$

$B := [d, e, f, ..., -i]$

$C := [a, b, c, ..., d, e, f, ...]$

where i and $-i$ are the opposite form terms and C consists of the terms in A and B except for the opposite form terms.

We want to show A and B imply C.

Recall we can imply a new clause using expansion by appending terms to an existing clause.

With this method, we can derive the following clauses:

$A' := [a, b, c, ..., d, e, f, ..., i]$

$B' := [a, b, c, ..., d, e, f, ..., -i]$

where A expands to A' and B expands to B'.

Now using reduction, it is seen that A' and B' imply C.

Since A expands to A' and B expands to B' and together A' and B' imply C, it can be said that A and B imply C. □

Contradicting clauses imply unsatisfiability

Claim: Contradicting 1- or 2-terminal clauses imply the instance is unsatisfiable.

Definition: A *k-terminal* clause is a clause which contains k terms.

Definition: Contradicting clauses are sets of clauses in which each clause consists of the same terms and there exists a clause for each combination

of positive or negative forms of these terms. For example, $[-a]$ and $[a]$ are contradicting 1-terminal clauses and $[a, b], [a, -b], [-a, b]$, and $[-a, -b]$ are contradicting 2-terminal clauses. Larger sets of contradicting clauses provide no use for the purposes of this paper.

Proof. First, we want to show contradicting 1-terminal clauses imply the instance is unsatisfiable.

Consider a 1-terminal clause containing the positive form of a terminal and another 1-terminal clause containing the negative form of that same terminal.

Recall a terminal can only be assigned one value, either True or False.

If the clause with the positive term is True, then the clause with the negative term is False and the instance is unsatisfiable.

If the clause with the negative term is True, then the clause with the positive term False and the instance is unsatisfiable.

Therefore contradicting 1-terminal clauses imply unsatisfiability.

Now we want to show contradicting 2-terminal clauses imply contradicting 1-terminal clauses.

Let the clauses be defined as follows:

$A := [a, b]$

$B := [a, -b]$

$C := [-a, b]$

$D := [-a, -b]$

where A, B, C, and D are contradicting 2-terminal clauses.

Notice we can derive the following clauses:

$AB := [a]$

$CD := [-a]$

where A and B imply AB and C and D imply CD.

These are contradicting 1-terminal clauses which have been shown to imply the instance is unsatisfiable.

Note that this derivation will always be possible because contradicting 2-terminal clauses contain all possible combinations of the same two terms in their positive and negative forms.

Therefore contradicting 1- and 2-terminal clauses imply the instance is unsatisfiable. □

Initial implication graph

Claim: If an instance is unsatisfiable, there exists an implication graph in which the given 3-terminal clauses expand to all possible n-terminal clauses which reduce to contradicting 1-terminal clauses.

Definition: An *implication graph* is a directed graph used to represent the path from one clause's existence to another clause's existence via the methods of Reduction, Expansion, or Implication. The nodes of the graph represent clauses and the edges represent the parent node(s) implying the child node.

Definition: This particular implication graph will be referred to as the *initial implication graph*. This will be used in contrast to other implication graphs which derive the same clauses, but process different intermediate clauses.

Definition: A *given 3-terminal clause* is a clause which is given as part of the original instance of the problem.

Definition: A *derived clause* is a clause which has been added to the instance to aid with further processing.

Definition: An *ancestor* of a clause, C, is any clause which was used in the implication of C.

Definition: A *descendant* of a clause, C, is any clause whose presence relies on an implication from C.

Proof. First, we will show that the given 3-terminal clauses can expand to every possible n-terminal clause. Next, we will show how these n-terminal clauses can reduce to contradicting 1-terminal clauses.

Recall an instance is unsatisfiable if there is no satisfying assignment.

This means all possible assignments are blocked.

For each blocked assignment, there exists a given 3-terminal clause which blocks that assignment.

For each blocked assignment, consider an n-terminal clause which blocks that assignment.

To construct the n-terminal clause, the terminals whose values are True will exist in their negative form and the terminals whose values are False will exist in their positive form.

In this way, each term in the clause is False and the assignment is blocked by that clause.

We want to show that every n-terminal clause can be derived by expanding a given 3-terminal clause.

Suppose not, then there exists an n-terminal clause to which no given 3-terminal clause can expand.

Recall all of the assignments are blocked.

Therefore there exists a given 3-terminal clause that blocks the same assignment as this n-terminal clause.

If the given 3-terminal clause cannot expand to this n-terminal clause (due to the fact that there are terms in the 3-terminal clause which do not exist in the n-terminal clause), then that 3-terminal clause cannot block that assignment.

This is a contradiction because that 3-terminal clause blocks that assignment.

Therefore the given 3-terminal clauses can expand to every possible n-terminal clause.

Now we want to show these n-terminal clauses can reduce to contradicting 1-terminal clauses.

Notice there exists an n-terminal clause containing every term for each possible combination of positive and negative forms of these terms.

Pick a terminal in the instance and look at the clauses which contain the positive form of that term.

Notice that for each of these clauses containing the positive form of the term, there exists a clause which is identical in every term except it contains the negative form of that term.

This is because each clause contains every term and there exists a clause for each combination of either form of each term.

Using the idea of Reduction, we can pop this term and imply a clause which contains the remaining terms.

Since the clauses before this reduction contained every possible combination of these remaining terms and we only popped this opposite form term, all of the clauses now have every possible combination of the remaining $n-1$ terms in either form.

It can be seen how this pattern will continue since we will always have every possible combination of the remaining terms at any step.

Once we get to the step which derives the 1-terminal clauses, there is only one term that remains and it must be positive in one clause and negative in the other.

Therefore the n-terminal clauses can reduce to contradicting 1-terminal clauses.

In total, an unsatisfiable instance implies that the given 3-terminal clauses can be expanded to n-terminal clauses which can then be reduced to contradicting 1-terminal clauses. □

Implication graph without expansion

Claim: The initial implication graph can be arranged in such a way as to derive contradicting 1-terminal clauses without expansion.

Definition: The 1-terminal clauses which are derived in this implication graph are sometimes referred to as the *final output* clauses of the implication graph.

Definition: This modified implication graph will be referred to as the *expansionless implication graph.*

Proof. This idea comes in two parts: First, we consider the 3-terminal to n-terminal expansion and the resulting n-1, n-2, ... reductions. We show how the opposite form terms in the reductions can be popped first and then the resulting clause can be expanded as opposed to the initial implication graph in which the clauses are expanded first and then the opposite form terms are popped. Second, we show how the expansion in the latter step is unnecessary to derive the final output of the implication graph.

Recall the shape of the initial implication graph: the 3-terminal clauses expand to n-terminal clauses which reduce to contradicting 1-terminal clauses.

For each n-terminal clause, there exists at least one corresponding given 3-terminal clause which expands to that n-terminal clause.

Let the clauses be defined as follows:

$A := [a, b, c]$

$B := [d, e, f]$

$A' := [a, b, c, ...]$

$B' := [d, e, f, ...]$

where A and B are given 3-terminal clauses which expand to the n-terminal clauses, A' and B', respectively. Additionally, A' and B' reduce to an n-1-terminal clause.

Consider the opposite form term which is popped from A'.

This term either exists in A or it does not exist in A (in the latter case, it is one of the expanded terms which are used to derive A').

If this term does not exist in A, then A can simply expand to this n-1-terminal clause since all of the terms in A exist in this clause.

If the term does exist in A, consider its corresponding opposite form term which exists in B'.

This term either exists in B or it does not exist in B.

If this term does not exist in B, then B can expand to this n-1-terminal clause since all of the terms in B exist in the clause.

If the term does exist in B, then A and B can imply a new clause (since they share an opposite form term).

This clause then expands to the n-1-terminal clause since all of the terms in this clause exist in that n-1-terminal clause.

So we can derive the n-1-terminal clauses by delaying expansion until the last step.

Using the principle of mathematical induction, we will show how expansion can be delayed until the last step for every clause in the implication graph.

Consider the k-terminal clause to k-1-terminal clause reduction and assume we have derived the k-1-terminal clauses by delaying expansion until the last step. Want to show we can derive the k-2-terminal clauses by delaying expansion until the last step.

The k-1-terminal clauses reduce to the k-2-terminal clauses say by popping the opposite form term, a.

The path from the given 3-terminal clauses to the k-1-terminal clauses went through zero or more implications or reductions before expanding to the k-1-terminal clauses.

Let the clauses C and D be the clauses in that implication graph right before the expansion and the clauses C' and D' be their respective k-1-terminal clauses right after the expansion.

The opposite form term, a, is in C', but it could either be in C or it could not be in C.

If it is not in C, then C can expand to the k-2-terminal clause since all of the terms in C exist in the k-2-terminal clause.

If it is in C, consider placement of the term $-a$ in D'.

The term $-a$ is in D', but it is either in D or it is not in D.

If it is not in D, then D can expand to the k-2-terminal clause since all of the terms in D exist in the k-2-terminal clause.

If it is in D, then C and D imply another clause, CD, (since they share an opposite form term) and that clause can expand to the k-2-terminal clause

since the opposite form term is popped and all of the terms in CD exist in that k-2-terminal clause.

Therefore we can always delay expansion until the last step in this implication graph.

Now we want to show that expansion is unnecessary when the entirety of the graph is taken into consideration.

Recall the final output of the implication graph consists of two 1-terminal clauses.

When working with strictly the methods of Reduction and Implication, you will not derive a clause with a length less than 1 (if you do, you have already derived contradicting 1-terminal clauses and have shown the instance is unsatisfiable).

Since the output is of length 1 and the shortest clause derived is of length 1, there is no need for expansion to derive the final output clauses. □

The shape of the implication graph

Claim: The expansionless implication graph can be rearranged such that it can be thought of as a single clause, *the running clause*, which grows and shrinks in size as it is used with given 3-terminal clauses to imply the next running clause. The output of this modified implication graph consists of contradicting 2-terminal clauses.

Definition: The *running clause* refers to a single clause in a series of clauses in which each running clause can derive the next using implication with a given 3-terminal clause. For example consider the following implications,

1. A and B imply AB
2. AB and C imply ABC
3. ABC and D imply $ABCD$

Here, A, B, C, and D are given 3-terminal clauses and each output is the running clause for that step. The important aspect of a running clause is each implication's inputs consist of at least one given 3-terminal clause and at most one derived clause. There will never be an implication which consists of two derived clauses.

Disclaimer: Rather than looking at the implication graph which derives contradicting 1-terminal clauses, it is more helpful to look at four distinct branches, each of which derive one of the four contradicting 2-terminal

clauses. Clearly the 2-terminal clause to 1-terminal clause implication requires two derived clauses as inputs (because 2-terminal clauses will never be given). To make full use of the running clause, we only concern ourselves with a single branch which derives one of these 2-terminal clauses. The same methods can then be repeated for the other three branches which can then be used to imply contradicting 1-terminal clauses.

Proof. The idea is to use the fact that every term in this implication graph exists in a given 3-terminal clause and those 3-terminal clauses can be used directly without having to process another derived clause first.

Since no new terms are introduced by way of expansion, the only place new terms are introduced are in the given 3-terminal clauses.

Notice, too, that every term in an output of an implication exists in at least one of the input clauses in that implication.

If there is ever an implication between two derived clauses, say A and B imply C, that pops some opposite form term, say $a \in A$ and $-a \in B$, then there is a given 3-terminal clause, D, which contains $-a$ and the implication can be done by using A and D to imply a new clause AD which does not contain a or $-a$.

There are two points to notice about this new implication: (1) there may be terms in AD which do not exist in C and (2) there may be terms in C which do not exist in AD.

Regarding point (1), we want to show the terms in AD can be popped in a very similar way as the terms in the implication graph for deriving B. Any descendant of AD in this implication graph will be referred to as AD' (and can be thought of as the running clause).

Consider the implication graph which derives B. Each clause in this graph pops a term and introduces up to two terms.

Since D is an ancestor of B, it introduces the very same terms to AD as it does in the implication graph of B.

Any term that is introduced to AD' is a term that exists in the implication graph of B and is popped just the same.

Recall the terms in the implication graph of B exist in given 3-terminal clauses so these terms in AD' can be popped with given 3-terminal clauses.

Notice the terms in AD' now consist of the terms in A and at least some of the terms in B.

All of the terms in AD' exist in C.

Regarding point (2), the terms in C exist in A or B or both. Clearly the terms in A which exist in C also exist in AD' so now we are only concerned with the terms in B that may not exist in AD'.

The implication graph of B pops terms that are not in B and introduces terms that are in B.

The clauses that introduce terms which stay in B are the same clauses which are used to pop terms from AD'.

Therefore these same terms are introduced to AD' just as they are to B.

The terms in B which exist in C also exist in AD'.

Since all the terms in C exist in AD' and all the terms in AD' exist in C, C can be derived just as AD' is derived.

C can be derived such that each implication's inputs contain at least one given 3-terminal clause and at most one derived clause. We refer to the derived clause in each step as the running clause.

Since each implication has at least one 3-terminal clause, the shortest possible output for the graph is a 2-terminal clause so there are now four branches resulting in contradicting 2-terminal clauses rather than two branches resulting in contradicting 1-terminal clauses. □

Limiting the max length

Claim: The implication graph centered around a running clause can be rearranged in such a way as to derive the final output contradicting 1-terminal clauses by processing clauses with a maximum length of 4.

Proof. Recall the implication graph is composed of a running clause that is used in conjunction with given 3-terminal clauses to imply a new running clause.

At each step, the running clause loses a term and gains up to two terms.

One important note is that if the next running clause is shorter, then the given 3-terminal clause input must have shared two terms with the previous running clause.

Additionally, if the running clause is not shorter, then any newly introduced term must be popped later in the implication graph.

Without loss of generality, consider the following description of the graph at this point:

1. $[a, b]$ is the final output
2. $[a, b, -c]$ is the final given 3-terminal clause used in the graph

These are always true because $[a, b]$ is the final output of the branch with which we are concerned and there must be some term which is popped in the very last step of this derivation. Let this term be c which is popped by the clause, $[a, b, -c]$.

It seems to make the most intuitive sense to start with the final clause and move up the chain of clauses until we derive the final output $[a, b]$.

We will use the principle of mathematical induction to derive the clause $[a, b, -i]$ (by processing clauses with a maximum length of 4) for every term, i, which is introduced to the running clause.

Starting with $[a, b, -c]$, we know the last running clause is $[a, b, c]$ because $[a, b]$ is the final output and c must be popped by a 3-terminal clause.

Now consider the second to last clause. Let the second to last popped term be d. The clause must have $-d$ as well as two terms from the following set: $\{a, b, c\}$. This is because, in order to be the second to last clause, it must contain only the popped term as well as term(s) from the running clause (if it contained anything else, it would require another step to pop the newly introduced terms).

We want to derive the clause $[a, b, -d]$.

There are three options for this second to last clause:

$[a, b, -d], [a, c, -d], [b, c, -d]$.

In the first case, we already have $[a, b, -d]$.

In the latter two cases, we know $[a, b, -c]$ exists so we can pop c and derive $[a, b, -d]$.

In general, consider the clause $[-x, y, z]$ where x is popped from the running clause and y and z are in the running clause.

We want to derive $[-x, a, b]$.

Since y and z are in the running clause, they must be a or b or popped later.

In the case where y or z are a or b, clearly these terms are not popped since they exist in the final output so disregard any reference to popping these terms in the following steps. The terms a and b will overlap and the resulting $[-x, a, b]$ clause will be the same in all cases where y and z are popped terms or they are a and/or b.

Consider y and z as popped terms, they must be popped later in the implication graph and since we're moving from last popped term to first popped term, we have already derived $[-y, a, b]$ and $[-z, a, b]$.

Now we can pop y to derive $[-x, z, a, b]$

and we can pop z to derive $[-x, a, b]$.

In this way, we can derive $[-x, a, b]$ by processing clauses with a maximum length of 4 for all x in which x is a popped term in the running clause.

Consider the part of the implication graph which introduces terms to be popped. The first clause in this graph cannot pop any terms since there are no terms in the running clause.

As such, all of the terms in the first clause are positive as they are introduced for the first time.

Recall we have constructed a clause, $[-x, a, b]$, for each term, x, which is popped from the running clause.

As such, we can pop all the positive terms from this first clause and we will be left with the clause $[a, b]$.

It is shown we can derive the final output of the running clause based implication graph, $[a, b]$, by processing clauses with a maximum length of 4.

This can be done for each branch and the resulting contradicting 2-terminal clauses can imply contradicting 1-terminal clauses. □

Algorithm

This section reiterates the algorithm and proves its time complexity and correctness.

The algorithm steps are outlined as follows:

1. For each clause in the instance, C:
    (a) For each clause in the instance, D, which is not C:
        i. If C and D are of length four or less and share an opposite form term, add a new clause to the instance which consists of each term from both clauses except for the opposite form term.
2. For each clause in the instance, E, of length 1:
    (a) For each clause in the instance, F, of length 1:
        i. If E and F contain opposite form terms, the instance is unsatisfiable.
3. Repeat steps 1. and 2. as long as at least one clause is added to the instance
4. When no new clauses are added, the instance is satisfiable.

Time Complexity

Let n be the number of terminals in the instance. Since the instance only contains clauses of up to length 4 and each clause contains either form of four unique terminals, iterating the instance takes a maximum of $\binom{2n}{4} + \binom{2n}{3} + \binom{2n}{2} + \binom{2n}{1}$ steps, which is on the order of $O(n^4)$.

Steps 1. 1a. 2. and 2a. iterate through the instance so they are all on the order of $O(n^4)$.

Step 1ai. is done in constant time since the length of the clauses have a fixed maximum of 4 terms.

Step 2ai. is done in constant time since both clauses are of length 1.

Step 3. adds at least one clause to the instance so it is repeated at most once per possible clause in the instance. There are $\binom{2n}{4} + \binom{2n}{3} + \binom{2n}{2} + \binom{2n}{1}$ possible clauses in the instance so this step is done on the order of $O(n^4)$ times.

Step 4. is done in constant time with the use of a flag or similar technique.

Steps 1. and 1a.. repeat each time step 3. is done.

Similarly steps 2. and 2a. repeat each time step 3. is done.

The worst case time complexity for each of these nested steps is $O(n^8)$ both of which are repeated $O(n^4)$ times by step 3.

This results in an overall time complexity of $O(n^{12})$.

Proof of Correctness

This algorithm is correct under the following conditions: an instance is unsatisfiable $\iff$ contradicting 1-terminal clauses can be derived using the algorithm.

The algorithm works by using the methods of Implication and Reduction which have been shown to only add clauses to the instance if those clauses do not block any assignments which were not already blocked.

As such, if the algorithm derives contradicting 1-terminal clauses, then all the assignments were already blocked and the instance was already unsatisfiable.

It is also seen that an unsatisfiable instance implies the existence of an initial implication graph in which the given 3-terminal clauses expand to n-terminal clauses which then reduce to contradicting 1-terminal clauses.

This initial implication graph can be rearranged such that no expansion step is necessary.

This expansionless implication graph can be rearranged again such that each implication consists of at least one given 3-terminal clause and at most one derived clause. The derived clause in each step is referred to as the running clause.

This running clause based implication graph can then be rearranged such that the contradicting 1-terminal clauses can be derived by processing only clauses with a maximum length of 4.

Taking all these methods into consideration, it is seen that processing an unsatisfiable instance using the idea of implication among clauses of length 4 or less results in the derivation of contradicting 1-terminal clauses.

Therefore, the algorithm derives contradicting 1-terminal clauses if and only if the instance is unsatisfiable.

Since this algorithm is done in polynomial time, 3SAT $\in$ P and P = NP.

References

[1] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.