October 1989

P Is Not Equal to *NP*

Jon Freeman
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-72.

P Is Not Equal to *NP*

Abstract

This paper presents a proof of the conjecture that the complexity classes *P* and *NP* are not equal. The proof involves showing that a particular problem in *NP* cannot be solved in polynomial time. The problem in question is the satisfiability problem for logical expressions in conjunctive normal form (*CSAT*). The strategy of the proof is to construct what amounts to the most efficient algorithm for *CSAT* and then show that this algorithm does not run in polynomial time. The algorithm is constructed by constructing, from first principles, a set of constraints that any efficient algorithm for *CSAT* must satisfy. The constraints are made so restrictive that any two algorithms that satisfy all of them are essentially interchangeable.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-72.

P IS NOT EQUAL TO NP

Jon Freeman

MS-CIS-89-72

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

Otober 1989

$\mathcal{P}$ Is Not Equal to $\mathcal{NP}$!

Jon Freeman
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

October 1989

Abstract

This paper presents a proof of the conjecture that the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ are not equal. The proof involves showing that a particular problem in $\mathcal{NP}$ cannot be solved in polynomial time. The problem in question is the satisfiability problem for logical expressions in conjunctive normal form *(CSAT)*. The strategy of the proof is to construct what amounts to the most efficient algorithm for *CSAT* and then show that this algorithm does not run in polynomial time. The algorithm is constructed by constructing, from first principles, a set of constraints that any efficient algorithm for *CSAT* must satisfy. The constraints are made so restrictive that any two algorithms that satisfy all of them are essentially interchangeable.

1 Introduction

Almost no one questions the assertion that the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ are not equal; the problem is to prove it. To prove this assertion, it suffices to prove that any particular problem in $\mathcal{NP}$ is not in $\mathcal{P}$, that is, *no* algorithm for that problem runs in polynomial time. Needless to say, a straightforward proof using this strategy is impossible, because the number of algorithms that can solve any particular problem is uncountably infinite. We have to use a different strategy.

Here is the strategy I propose. First, choose a particular problem in $\mathcal{NP}$. Then, *by reasoning from first principles*, construct a set of constraints such that any algorithm that solves that problem efficiently must satisfy every constraint in the set. Make the constraints so restrictive that any two

algorithms that satisfy them are essentially interchangeable. Construct an algorithm A that satisfies the constraints and argue that any other algorithm that does the same is equivalent to A in this sense. Finally, demonstrate that A does not run in polynomial time. Since the problem is in $\mathcal{NP}$, it follows that $\mathcal{P} \neq \mathcal{NP}$.

Clearly, the success of this strategy depends critically on the nature of the problem. Most non-trivial problems do not lend themselves to this strategy; if they did, then the study of algorithms would certainly be less challenging.

The problem that I choose for my proof is the satisfiability problem for logical expressions in conjunctive normal form, known as *CSAT*. The following section briefly describes *CSAT*.

2 A Brief Description of *CSAT*

A *Boolean variable* (or *logical variable*) is a variable that can be either true or false (denoted by **T** and **F**, respectively). A *disjunction* is a sequence of one or more logical expressions separated by the Boolean *or* operator ($\vee$); a *conjunction* is a sequence of one or more logical expressions separated by the Boolean *and* operator ($\wedge$). A *literal* is either a Boolean variable or the complement of one. A *clause* is a disjunction of literals. A logical expression is in *conjunctive normal form (CNF)* if it is a conjunction of clauses. For example, the following expression is in CNF:

$$(P \vee R \vee \overline{S}) \wedge (\overline{Q} \vee \overline{R}) \wedge (\overline{P} \vee S) \wedge (Q \vee \overline{R} \vee S)$$

where P, Q, R, and S are Boolean variables (hereafter referred to as variables).

A *truth assignment* is a function from the set of logical variables to {**T**, **F**}. Given a truth assignment h and a logical expression E, h *satisfies* E if h makes E true; otherwise, h does not satisfy E. We also say that a truth assignment h *satisfies* a clause C or a literal L if h makes C or L true, respectively.

The CNF-satisfiability problem *(CSAT)* is defined as follows: Given a logical expression E in CNF, is there a truth assignment h that satisfies E? Note that *CSAT* is a *decision problem*, that is, a problem that requires a yes or no answer.

3 Constraints, Unconstraints, and Facts

I begin my proof by listing the constraints that must hold for any efficient algorithm A that solves $CSAT$ for a given CNF-expression E in the manner described above. I also list some *unconstraints* on A, which specify choices that A has to make that do *not* affect the amount of time A takes to find a solution. Finally, I list some *facts* that help determine some of these constraints and help justify others.

3.1 Initial Constraints

Constraint 1 *A terminates when it has found a truth assignment that satisfies E, or when it has determined that no such truth assignment exists.*

Justification: This is what it means for A to solve $CSAT$ for E.

Constraint 2 *A satisfies E by satisfying every clause in E.*

Constraint 3 *A satisfies a clause in E by satisfying at least one literal in that clause.*

Justification: These two constraints follow from what it means for a Boolean expression to be true.

Fact 1 *During the execution of A, every variable in E can have one of three possible truth values: true, false, or indeterminate. Initially every variable is indeterminate. The same is true for literals.*

Fact 2 *During the execution of A, every clause C in E can have one of three possible truth values: true (at least one literal in C is true), false (all literals in C are false), or indeterminate (out of a total of n literals in C, k are false, where $0 \leq k \leq n-1$; the rest are indeterminate). Initially every clause is indeterminate.*

Fact 3 *During the execution of A, E itself can have one of three possible truth values: true (all clauses in E are true), false (at least one clause in E is false), or indeterminate (out of a total of n clauses in E, k are true, where $0 \leq k \leq n-1$; the rest are indeterminate). Initially E is indeterminate.*

Constraint 4 *When trying to satisfy E, A only tries to satisfy indeterminate clauses in E, and when trying to satisfy an indeterminate clause C, A only tries to satisfy indeterminate literals in C.*

Justification: This should be obvious.

Constraint 5 *If A tries to satisfy an indeterminate clause C and A succeeds in doing so by satisfying one of its indeterminate literals, A must not try to satisfy any additional indeterminate literals in C.*

Justification: This is all that is necessary; trying to satisfy any additional indeterminate literals would be a waste of time.

3.2 Adding Parallelism to A

It is sometimes possible to add parallelism to A.

Fact 4 *If we partition the set of clauses in E into blocks (i.e., subsets) of clauses such that no two blocks have any variables in common, then we can satisfy those blocks independently, and hence in parallel.*

Justification: The truth assignments for each block do not interfere with each other, because they range over disjoint sets of variables.

The next two facts should both be obvious.

Fact 5 *If we partition the set of clauses in E into blocks having the above property, and we manage to find a satisfying truth assignment for each block, then the satisfying truth assignment for E is just the composition of the satisfying truth assignments for each block.*

Fact 6 *If we partition the set of clauses in E into blocks having the above property, and we determine that at least one of these blocks is a contradiction, then E itself is a contradiction.*

Fact 7 *We can partition the set of clauses in E into blocks having the above property in polynomial time.*

Justification: This problem reduces to the problem of finding the connected components of a graph. Given E, we can construct an appropriate graph G with n vertices and m edges as follows. There is one vertex in G for every *possible* literal in E, that is, every variable that appears in E and its complement. There is one edge between every pair of vertices whose corresponding literals belong in the same block. Thus there is an edge between every pair of vertices corresponding to a literal and its complement, and for

every clause in E containing k literals, there are $k-1$ edges linking those corresponding vertices together. Let r be the number of variables that appear in E, and let s be the total number of literals in E. Then n is equal to $2r$, and m is equal to $r+(s-1)-$the total number of clauses in E; therefore m is bounded above by $r+s$.

We can construct G in time proportional to s, and we can find the connected components of G in time $\Theta(\max(n,m))$, which is usually just $\Theta(m) = O(r+s)$. Therefore we can find the blocks of E in time $O(r+s)$.

Constraint 6 *A must partition the set of clauses in E into blocks having the above property and then try to satisfy those blocks in parallel.*

Justification: With parallelism, the running time of A is a function of the number of clauses in the largest block; without parallelism, the running time is a function of the total number of clauses in E, which is always greater than or equal to the number of clauses in the largest block.

The constraints in the next two sections only apply to the operation of A with respect to a block of clauses B, not to E itself. In these sections, the phrase "the current truth assignment" refers to the current truth assignment for B.

3.3 Initial Constraints for Blocks of Clauses

Constraint 7 *A must not change the current truth assignment (i.e.,* backtrack*) until it absolutely has to.*

Justification: A must assume that the current truth assignment is correct until it has definite evidence to the contrary; otherwise, it will not be finding a satisfying truth assignment as quickly as possible.

Fact 8 *There are 2^n possible ways in which A can extend the current truth assignment by n variables (called* candidate extensions*).*

Justification: This should be obvious.

Constraint 8 *If a candidate extension falsifies at least one clause in B, then A must detect this problem and fix it.*

Justification: This follows from Constraint 2.

Constraint 9 *If a candidate extension falsifies at least one clause in B, then A must detect this problem and fix it* immediately.

Justification: If A continues to extend the current truth assignment after a candidate extension falsifies at least one clause in B, it may have to undo some of this work when it eventually gets around to fixing the false clause(s), thereby wasting time. For example, consider the following expression.

$$(P \vee Q) \wedge (R \vee S) \wedge (\overline{P} \vee \overline{R}) \wedge (T \vee U) \wedge (V \vee W) \wedge (\overline{Q} \vee \overline{T} \vee \overline{V}) \wedge (\overline{S} \vee \overline{T} \vee \overline{V})$$

Suppose that A tries to satisfy this expression as follows. First, A extends the (empty) truth assignment by the variables P and R, where both variables are true. This extension falsifies the third clause, but before A tries to fix this problem, it extends the truth assignment again by the variables T and V, where both variables are true. Now there is no way that A can fix the third clause without undoing part of the second extension. In order to make the third clause true, A must either make P false, R false, or both P and R false. If A makes P false, it must make Q true to satisfy the first clause. If A makes R false, it must make S true to satisfy the second clause. If A makes both P and R false, it must make both Q and S true. If A makes Q true, however, the second-to-last clause will become false; if A makes S true, the last clause will become false; and if A makes both Q and S true, both of these clauses will become false. The only way to fix these problems is to change the truth values of T, V, or both T and V respectively. In any case, part of the second extension is wasted.

Constraint 10 *Suppose that A is trying to extend the current truth assignment by n variables at the same time. A must keep trying each of the 2^n candidate extensions in turn until either it finds a candidate extension that does not falsify at least one clause in B or it runs out of candidate extensions to try.*

Justification: In the above situation, suppose that A tries one of the 2^n candidate extensions, and that candidate extension falsifies at least one clause in B. By Constraint 9, A must do something about this problem immediately. There are only two other things that A can do:

1. Give up prematurely and change the current truth assignment. According to Constraint 7, this is inefficient.

2. Keep the current truth assignment as it is, but try to extend it by a different set of variables (i.e., different with respect to either the number of variables, the variables themselves, or both). If A does this, then it will not have gained any information from the candidate extension, so it will have wasted time.

Therefore A must pick another candidate extension and try again.

The next two facts should both be obvious.

Fact 9 *In the above situation, if A runs out of candidate extensions to try, then the proposed extension does not work.*

Fact 10 *In the above situation, if A runs out candidate extensions to try and the current truth assignment is empty, then B is a contradiction.*

The following constraint is what I am really after.

Constraint 11 *A extends the current truth assignment by exactly one variable at a time.*

Justification: Suppose that A is trying to extend the current truth assignment by n variables at the same time, where $n \geq 1$. Further suppose that this extension does not work. By Constraint 10, A will try all 2^n candidate extensions in turn. A will determine that the extension does not work, but it will take time proportional to 2^n to do it. Clearly, A must minimize the amount of time it takes to make this determination; the only way to do this is to choose $n = 1$.

This constraint is critically important because it tells us that, although we can parallelize the operation of A with respect to E, we cannot parallelize it with respect to B. In other words, we cannot add any more parallelism to A.

Constraint 12 *A tries to satisfy exactly one clause in B at a time.*

Justification: This follows trivially from the previous constraint, because satisfying a clause involves extending the current truth assignment.

3.4 Backtracking

Constraint 13 *If, given the current truth assignment, at least one clause in B is impossible to satisfy, then A must detect this problem and fix it by backtracking.*

Justification: This follows from Constraint 2.

Constraint 14 *If, given the current truth assignment, at least one clause in B is impossible to satisfy, then A must detect this problem and fix it* as soon as possible *by backtracking.*

Justification: Suppose that A is trying to satisfy a particular clause that is impossible to satisfy, and A gives up after only trying to satisfy a few of its indeterminate literals. After A does some additional work, it will eventually try to satisfy the clause again and discover that it cannot do so; then it will backtrack, at which point it may undo some of the additional work, thereby wasting time.

Constraint 15 *A must successively try every possible way to satisfy a particular clause.*

Justification: This follows from the previous constraint, because the clause may be impossible to satisfy.

Constraint 16 *Suppose that A is trying to satisfy a clause C by picking an indeterminate literal L in C and making L true. If the resulting truth assignment falsifies at least one other clause in B, A must change it to make L false and then check whether* this *truth assignment falsifies at least one other clause in B.*

Justification: This follows from Constraint 10 with $n = 1$.

Constraint 17 *Again suppose that A is trying to satisfy a clause C by picking an indeterminate literal L in C and making L true. If A encounters an immediate problem, and then makes L false and encounters no immediate problem, A must immediately pick another indeterminate literal in C and try to satisfy that literal instead.*

Justification: This is what it means for A to successively try every possible way to satisfy C.

Constraint 18 *In the above situation, if A runs out of indeterminate literals in C to satisfy, it must backtrack immediately.*

Justification: There is no other way to satisfy C, so Constraint 14 applies.

Unconstraint 1 *The order in which A tries to satisfy the indeterminate literals in an indeterminate clause does not matter.*

Justification: We have no prior information about the indeterminate literals, other than the fact that they are indeterminate, that could help us determine an appropriate order.

Constraint 19 *Again suppose that A is trying to satisfy a clause C by picking an indeterminate literal L in C, etc. If A makes L both true and false, and both of the resulting truth assignments falsify at least one other clause in B, then A must backtrack immediately.*

Justification: B may not be a contradiction, and given the current truth assignment, there may not be a way to satisfy B that does not require making L either true or false. A must backtrack immediately because, if this is in fact the case, A will waste time trying to find a way to satisfy B without using L. For example, consider the following expression.

$$(P \vee Q) \wedge (R \vee S) \wedge (T \vee U) \wedge (V \vee W) \wedge (\overline{P} \vee \overline{R} \vee V) \wedge (\overline{R} \vee \overline{T} \vee \overline{V})$$

By Unconstraint 1, we can assume without loss of generality that A tries to satisfy the indeterminate literals in a clause from left to right. Let us also assume for now that A satisfies the clauses in the expression from left to right. (I will justify this assumption in Section 3.8.) A proceeds by making P true, then R true, then T true. If A then makes V true, the last clause becomes false; if A makes V false, the second-to-last clause becomes false. If A ignores V and makes W true, it will be unable to satisfy either of the last two clauses. This expression is not a contradiction, however; for example, A can satisfy it by making P true, R false, S true, T true, and V true. Therefore A must backtrack.

Constraint 20 *A must keep track of the sequence of clauses in B that it has explicitly satisfied.*

Justification: Otherwise, if A has to backtrack, it will not be able to determine which clause it should backtrack to.

I will address the issue of exactly *which* clause A should backtrack to in Section 3.6.

Constraint 21 *When A backtracks from a clause C to a clause C', A must retract all the assignments of truth values to variables that it made when trying to satisfy both C and C'.*

Justification: The whole idea behind backtracking is to find a different way to satisfy C', and since A satisfied C' before it satisfied C, A must also retract the changes it made to the truth assignment while trying to satisfy C.

The next section applies to E itself, not just to a block of clauses in E.

3.5 One-Literal Clauses

I now consider the special status of clauses in E that contain exactly one literal.

Fact 11 *The variables that appear in one-literal clauses in E are immune from backtracking. No other variables in E are immune from backtracking.*

Justification: There is only one way to satisfy a one-literal clause. There is always more than one way to satisfy a clause of length greater than or equal to 2.

Constraint 22 *If a variable in E is immune from backtracking, A must recognize it and treat it as such.*

Justification: It is clearly a waste of time to retract the truth value of a variable that can only have one truth value.

This means that when A backtracks past a clause C, A must retract the truth values of all the variables in C *except* the variables that are immune from backtracking. (See Constraint 21.)

Constraint 23 *A must assign truth values to the variables that are immune from backtracking* first.

Justification: If a clause C of length at least 2 contains an indeterminate literal L that is the same as some one-literal clause C', then satisfying C' will satisfy L, and hence C, so it would be inefficient to try to satisfy another indeterminate literal in C, as doing so might very well lead to a backtracking, but satisfying C' by definition will not.

Therefore all of the previous constraints only apply to clauses of length at least 2.

Unconstraint 2 *The order in which A satisfies the one-literal clauses in E does not matter.*

Justification: This should be obvious.

We also have a necessary condition for testing whether E is a contradiction.

Fact 12 *If both a variable and its complement are one-literal clauses in E, then E is a contradiction.*

Note that this test only detects a few contradictions. In the next section, I will present a test that detects all other contradictions.

3.6 Overall Search Strategy

The next three sections only apply to a particular block of clauses B, not to the entire expression E.

We can think of a block B as defining a search tree, and we can think of the operation of A as a movement from the root of this tree to its leaves. Each node in the tree corresponds to a clause of B. The branches from each node correspond to the indeterminate literals of the appropriate clause. A branch emanating from a node corresponds to an indeterminate literal in the appropriate clause that, when satisfied, does not cause any immediate problems. Moving from one node to another along a given branch corresponds to satisfying the first node's corresponding clause by satisfying the appropriate literal and then attempting to satisfy the second node's corresponding clause.

We should therefore ask what search strategy A should adopt to move through this search tree. The strategy must satisfy the following two constraints:

1. It must move from one end of the search tree to the other as quickly as possible.

   Justification: This is what it means for A to find a satisfying truth assignment as quickly as possible.

2. It must be exhaustive.

   Justification: A must terminate on all inputs, so A must terminate on all contradictions, so A must systematically try all possible ways to satisfy the clauses of B to determine whether B is a contradiction.

The *only* search strategy that satisfies these two constraints is depth-first search. Therefore A must conduct a depth-first search of the search tree, so A itself must also satisfy these two constraints:

Constraint 24 *The order in which A satisfies the clauses in B must be* fixed.

Constraint 25 *When A backtracks past a clause in B, it must backtrack to that clause's immediate predecessor in the clause sequence for B.*

We now have a means of determining whether B is a contradiction that succeeds when the test in the previous section fails.

Fact 13 *If A attempts to backtrack past the first clause in the clause sequence for B, then B is a contradiction.*

3.7 Failed Literals

All of the constraints in this section follow indirectly from the conclusions about A's overall search strategy presented above.

Constraint 26 *Suppose that A backtracks from a clause C to a clause C'. There is some literal L in C' that A explicitly satisfied in order to satisfy C'. A must not satisfy this literal again; it is a* failed literal, *i.e., a literal that, when satisfied, eventually led to an unsatisfiable clause.*

Justification: Satisfying such a literal twice might lead to an infinite loop, which is intolerable; besides, the whole idea behind backtracking is to find a different way to satisfy C'.

Constraint 27 *Along with the sequence of clauses in B that A has explicitly satisfied, A must also remember the literal in each of those clauses that it explicitly satisfied.*

Justification: A must remember which literal in a clause it explicitly satisfied so it can make that literal a failed literal if it ever backtracks to that clause. A cannot rely on the current truth assignment to identify this literal, because there may be other literals in the clause that are also true as a result of A's attempts to satisfy other clauses.

Constraint 28 *Suppose that A is trying to satisfy a clause C by picking an indeterminate literal L in C and trying to satisfy L. If L is a failed literal, A must make L false and then check whether* this *truth assignment creates an immediate problem.*

Justification: This constraint is true for the same reason that Constraint 16 is true. The only difference is that in in this case, we know that satisfying L leads to a long-term problem, as opposed to a short-term problem.

Constraint 29 *Again suppose that A is trying to satisfy a clause C by picking an indeterminate literal L in C, etc. If L is a failed literal, and making L false makes some other clause in B false, A must backtrack immediately.*

Justification: This constraint is true for the same reason that Constraint 19 is true; the only difference is that here, we know that satisfying L leads to a long-term problem, as opposed to a short-term problem. For example, consider the following expression.

$$(P \vee Q) \wedge (R \vee S) \wedge (T \vee U) \wedge (V \vee W) \wedge (\overline{P} \vee \overline{R} \vee V) \wedge (\overline{R} \vee \overline{T} \vee \overline{V}) \wedge (\overline{P} \vee \overline{R} \vee T)$$

By Unconstraint 1, we can assume without loss of generality that A tries to satisfy the indeterminate literals in a clause from left to right. Let us also assume for now that A satisfies the clauses in the expression from left to right. (I will justify this assumption in Section 3.8.) A proceeds by making P true, then R true, then T true. It then tries to make V both true and false and fails, so it backtracks to the third clause and makes T a failed literal for that clause. It tries to satisfy the third clause again; it discovers that T is a failed literal, and makes it false, but doing so makes the last clause false. If A ignores T and makes U true, it will be unable to satisfy the last clause. This expression is not a contradiction, however; for example, A can satisfy it by making P true, R false, S true, T true, and V true. Therefore A must backtrack again.

Fact 14 *A clause can have more than one failed literal.*

Justification: A may backtrack to the same clause over and over again, explicitly satisfying a different indeterminate literal each time. It can do this as many times as there are indeterminate literals in the clause.

Constraint 30 *Again suppose that A is trying to satisfy a clause C by picking an indeterminate literal L in C, etc. If L is a failed literal and making L false does not cause any immediate problem, A must immediately pick another indeterminate literal in C and try to satisfy that literal instead.*

Justification: Like Constraint 17, this follows from Constraint 15.

Constraint 31 *In the above situation, if A runs out of indeterminate literals in C to satisfy, it must backtrack immediately.*

Justification: There is no other way to satisfy C, so Constraint 14 applies.

Constraint 32 *When A backtracks from a clause C to a clause C', A must forget about all of the failed literals in C.*

Justification: The failed literals for a clause are only useful when A backtracks *to* that clause, not when A backtracks *past* it.

Constraint 33 *When A backtracks from a clause C to a clause C', A must also forget about the literal in C that it explicitly satisfied.*

Justification: This follows from Constraint 21.

3.8 Choosing an Initial Clause Order

Next, I consider the important issue of the order in which A should try to satisfy the clauses in B. Given no other constraints, we would conclude that A should use the following strategy: try to satisfy the clauses in B having the *smaller* number of indeterminate literals first. This is because, after satisfying these clauses, the number of indeterminate literals in each of the remaining clauses will be as large as possible, which maximizes the chance that A will be able to find indeterminate literals in those clauses that work, and thus minimizes the chance that A will backtrack. This implies that A should choose the clauses it satisfies *dynamically*; that is, A should find the clause in B having the smallest number of indeterminate literals, satisfy that

clause, and repeat. We have already concluded, however, that the order in which A satisfies the clauses must be fixed. Therefore A must choose some order in which to satisfy them at the start. What should this order be? The only prior information we have that could help us answer this question is the length of the clauses. We therefore have the following constraint.

Constraint 34 *If some of the clauses in B are of different lengths, A must try to satisfy the shorter clauses first.*

Justification: This follows from the approximation that clauses with fewer literals are more likely to have fewer *indeterminate* literals at any time during the execution of A.

Therefore a straightforward and obvious way (but certainly not the only way) to satisfy B that satisfies this constraint is to sort the clauses in B by increasing length and then satisfy those clauses from left to right. This approach also ensures that A satisfies all of the one-literal clauses in B first.

Unconstraint 3 *If some of the clauses in B are of the same length, A can try to satisfy them in any order.*

Justification: Again, we have no prior information about the clauses, other than their length, that could help us determine an appropriate order.

3.9 On the Efficiency of Data Structures

It may seem that the data structures for A should be as efficient as possible. This is not necessarily true, however, for the following reason. We are trying to determine whether or not A runs in polynomial time. A must satisfy all of the other constraints in this paper in order for us to make this determination. With respect to the data structures, though, what matters is not how efficient they are with respect to the operations they support, but *how many times they must perform those operations.*

As long as the data structures perform all the operations they support in polynomial time, the other constraints in this paper will determine how many times they must do so, and hence whether or not A runs in polynomial time. Therefore we have the following (easily satisfied) constraint.

Constraint 35 *The data structures for A must perform all of the operations they support in polynomial time.*

3.10 One Final Constraint

The last constraint is important for the purposes of the proof.

Constraint 36 *A must not do anything other than try to find a satisfying truth assignment for E, or indicate that no such assignment exists if it cannot do so.*

Justification: Anything else would be a waste of time.

4 On the Equivalence of Two Algorithms

When most people state that two algorithms are equivalent, they mean that, given the same inputs, the algorithms return the same outputs. This definition says nothing about the relationship of the time complexity of the algorithms, and is therefore too weak for my purposes. I would like to define equivalence in such a way that two algorithms are equivalent if, given the same inputs, they take exactly the same amount of time to find a solution.

Unfortunately, it is impossible to *formally* compare two algorithms to decide whether they are equivalent in this sense. The reason lies in the definition of an algorithm. An algorithm is defined as a description of a sequence of steps that solves a given problem. There is no constraint on the language of the description: it could be a formal language, an informal language, or even a natural language, as long as anyone who reads the description can understand it. Hence there is no place for a rigorous proof of equivalence, given such a definition. Instead, I define the equivalence of two algorithms in the following way.

Definition 1 *Two algorithms are* equivalent *if their overall sequence of atomic steps is the same and the actions they take during each atomic step (i.e., the basic operations they perform on data and decisions they make) are the same, irrespective of the way in which their descriptions are structured as procedures and irrespective of any aspects of either algorithm that do not affect the amount of time they take to find a solution (i.e., unconstraints).*

Claim 1 *This definition makes sense.*

5 An Efficient Algorithm for *CSAT*

This algorithm is called *A-CSAT*. I have written it in a list-processing language based on the programming language Scheme. In fact, the description that appears below is actually the heart of a complete, fully working Scheme program (with the exception of parallelism). Any assumptions the algorithm makes are also described below.

```
; Representations:
;
; A (Boolean) variable is represented as an atom.
; A literal is represented as either a variable or '(not V), where V is a
; variable.
; A clause is represented as a list of literals.
; A CNF-expression is represented as a list of clauses.

; We need a global list that contains all the one-literal clauses in the
; expression.

; For each block, we need a stack that contains all of the clauses of
; length at least 2 that the algorithm has explicitly satisfied.  Each
; entry in the stack is a pair of the form '(clause literal), where
; "literal" is the literal that the algorithm made true in order to make
; "clause" true.  We need to know what this literal is in case we have to
; backtrack over the clause; if we do, then it becomes a "failed literal"
; for the clause (see below).

; For each block, we also need a table that contains, for each clause
; in the expression of length at least 2, the literals within that clause
; which eventually led to a backtracking (known as "failed literals").
; These literals can not and should not be considered again.

; Variables, literals, and clauses can have one of three possible
; truth values:  true, false, and indeterminate, represented by 'true,
; 'false, and 'indeterminate, respectively.

; Each block has its own truth assignment, represented by a list of
; pairs of the form '(variable truth-value).

; The satisfaction procedure itself.  In this procedure, (comap f l) is
; an imaginary function that takes a function f and a list l, applies
; f to the elements of l in parallel, and returns the resulting list.
; If "map" is substituted for "comap", this procedure works just fine on
; a sequential machine.

(define (satisfy exp)
  (let ((sorted-exp (merge-sort exp)))
```

```
    (set! one-literal-clauses (one-literals sorted-exp))
    (let ((one-lit-h (satisfy-one-lits)))
      (if (equal? one-lit-h 'contradiction)
          (display "This expression is a contradiction.")
          (let ((big-sorted-clauses (two-or-more-lits sorted-exp)))
            (cond ((null? big-sorted-clauses)
                   (display
                    "There is a satisfying truth assignment:")
                   (newline)
                   (display one-lit-h))
                  (else
                   (let ((final-h
                          (flatten
                           (comap (lambda (block)
                                    (satisfy-block
                                     one-lit-h
                                     empty-failed-lits-table
                                     empty-pair-sequence
                                     block
                                     (first block)))
                                  (find-blocks big-sorted-clauses)))))
                     (cond ((member 'contradiction final-h)
                            (display
                             "This expression is a contradiction."))
                           (else
                            (display
                             "There is a satisfying truth assignment:")
                            (newline)
                            (display final-h)))))))))))

; The procedure for satisfying a block.  This procedure returns a
; satisfying truth assignment for the block, or '(contradiction) if
; the block is a contradiction.  The indeterminate clauses in the
; block are satisfied from left to right.

(define (satisfy-block h flt ps exp current-clause)
  (if (true-expression? h exp)
      h
      (let ((result (satisfy-clause h flt ps exp current-clause)))
        (if (equal? (first result) 'failed)
            (if (null? ps)
                '(contradiction)
                (let ((previous-clause (first (first ps)))
                      (previous-lit (second (first ps))))
                  (satisfy-block
                   (retract-all-lits h previous-clause)
                   (erase-failed-literals
```

```
                    (add-failed-literal
                     flt
                     previous-lit
                     previous-clause)
                    current-clause)
                   (previous-pairs ps)
                   exp
                   previous-clause)))
             (satisfy-block
              (second result)
              flt
              (third result)
              exp
              (find-next-clause
               (second result)
               (rest-of (member current-clause exp))))))))

; The procedure for satisfying an indeterminate clause.  The indeterminate
; literals in a clause are tried from left to right.  This procedure
; returns a list of three items:  a flag indicating success or failure,
; the resulting truth assignment, and the resulting pair-sequence.

(define (satisfy-clause h flt ps exp current-clause)
  (define (clause-loop h temp)
    (cond ((null? temp) (list 'failed h ps))
          ((indeterminate-literal? h (first temp))
           (if (failed-literal? flt (first temp) current-clause)
               (if (false-expression?
                    (extend-lit h (first temp) 'false)
                    exp)
                   (list 'failed h ps)
                   (clause-loop
                    (extend-lit h (first temp) 'false)
                    (rest-of temp)))
               (if (false-expression?
                    (extend-lit h (first temp) 'true)
                    exp)
                    (if (false-expression?
                         (extend-lit h (first temp) 'false)
                         exp)
                        (list 'failed h ps)
                        (clause-loop
                         (extend-lit h (first temp) 'false)
                         (rest-of temp)))
                    (list 'succeeded
                          (extend-lit h (first temp) 'true)
                          (push-pair (list current-clause (first temp))
```

```
                                    ps)))))
        (else (clause-loop h (rest-of temp)))))
(clause-loop h current-clause))
```

6 The Remainder of the Proof

The remainder of the proof is straightforward.

Claim 2 *Any efficient algorithm for CSAT satisfies all of the constraints in Section 3.*

Justification: We must show that it is not possible for an algorithm to fail to satisfy one or more of the constraints in Section 3 and still be more efficient than an algorithm that satisfies all of them.

There are two kinds of constraints in Section 3: trivial constraints and non-trivial constraints. The trivial constraints deal with the basic operation of A; every valid algorithm for *CSAT* must satisfy all of them. The non-trivial constraints are the ones that an algorithm for *CSAT* may or may not satisfy. The non-trivial constraints are these: Constraints 5–7, 9–12, 14–19, and 22–36.

The proof of this claim lies in the justifications of the non-trivial constraints. In each of these justifications, I have shown either directly or indirectly that if A fails to satisfy the constraint in question, its running time will either increase or remain the same at best (but this is never guaranteed). Therefore any efficient algorithm for *CSAT* must satisfy all of the non-trivial constraints, and hence all of the constraints, in Section 3.

Claim 3 *A-CSAT satisfies all of the constraints in Section 3.*

Justification: It was constructed that way.

Claim 4 *Any algorithm that satisfies all of the constraints of Section 3 is equivalent to A-CSAT.*

Justification: This claim follows from the restrictiveness of the constraints. The first 35 constraints are so restrictive that any algorithm that satisfies them must perform the same overall sequence of atomic steps and the same basic actions in each atomic step as *A-CSAT*. Constraint 36 guarantees that such an algorithm does not perform any other random steps.

These three claims imply that any efficient algorithm for *CSAT* is equivalent to *A-CSAT*. Therefore we can say that *A-CSAT* is the most efficient algorithm for *CSAT*.

Claim 5 *A-CSAT does not run in polynomial time.*

Justification: We can easily prove this with a simple counterexample.

Therefore no efficient algorithm for *CSAT* runs in polynomial time, so *no* algorithm for *CSAT* runs in polynomial time, so *CSAT* is not in $\mathcal{P}$. Since *CSAT* is in $\mathcal{NP}$, it follows that $\mathcal{P} \neq \mathcal{NP}$. □

Acknowledgments

I would like to thank Professors Sanguthevar Rajasekaran and Lokendra Shastri for their comments and criticisms.