Exploring Strategies for Performing Extended Resolution as a Preprocessing Step for SAT-Solving

Eliot Ball

May 24, 2015

Summary

Prior research in the field of logic has shown that the *extended resolution* proof system can facilitate polynomial-length proofs of propositions that require exponential-length proofs in the *resolution* proof system. Based on this, some research has been carried out within the field of software and hardware verification into using extended resolution as a part the procedure followed by a SAT-solver, with varying success. I explore, through the use of *genetic algorithms*, whether extension can be used effectively as a preprocessing step that removes redundancy prior to running a SAT-solver. Because of their important practical applications, I focus on formulas obtained from software verification problems.

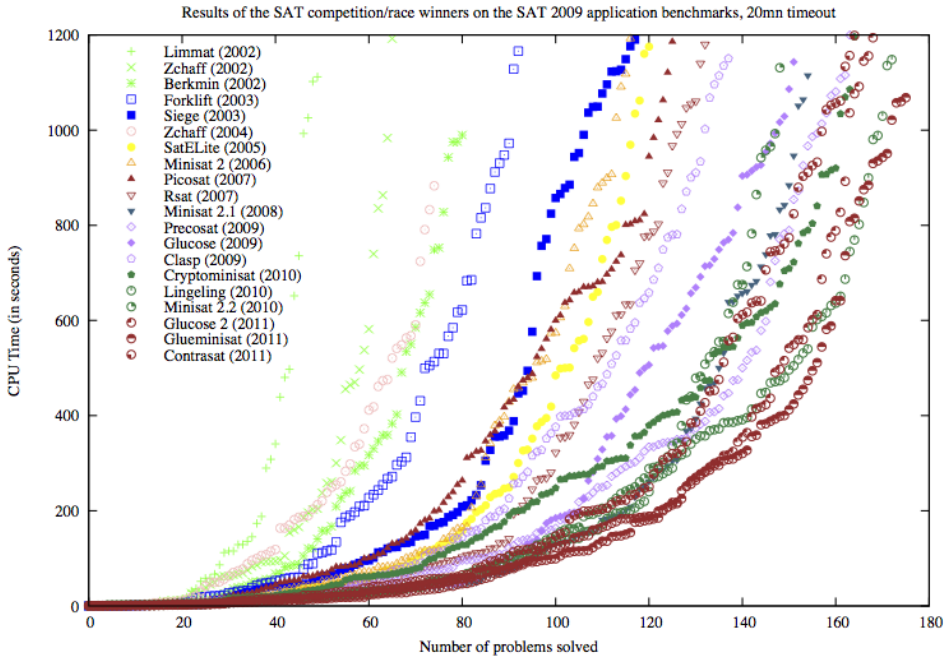
Contents

1 Introduction

In a world increasingly driven by machines and other automated systems, bugs and mistakes made by the creators of these machines are expensive, and sometimes dangerous. In 1994, Intel was pressured into replacing a huge batch of its Pentium processors when it was discovered that the floating point unit within them was faulty, often returning inaccurate results. This recall cost Intel $475 million [Nicely, 2008]. This type of mistake is often a huge problem for the parties involved, and it is easy to imagine truly disastrous outcomes if a mistake is made in the software for a missile guidance system, or the control systems for a car.

This raises the question of how to be sure that a system is correct — answering the question, "have we implemented what we intended to implement?" Testing a system, by trying out all of the different types of situations we think it can reach, can only go so far. What if we fail to foresee some of the situations? Ideally, we would be able to prove beyond doubt that the system does not fail.

There are several approaches to this, each with their own strengths and weaknesses [D'silva et al., 2008]. Bounded Model Checking (BMC) is one approach, where a system is encoded as a propositional formula that can be thought of as expressing the claim "There is a counterexample, proving this system flawed, of length at most k", for some given k [Biere et al., 1999]. This formula is then checked to see whether it is satisfiable using a SAT-solver. If the formula is satisfiable, then there is a counterexample of the desired length, which may be obtained by inspecting the values of the variables in a satisfying assignment. If the formula is not satisfiable, then we can be sure that there is not a counterexample of the desired length. The smallest possible k guaranteeing that the system is completely correct (that there is no counterexample of any length) is called the *completeness threshold* [Kroening and Strichman, 2003]. In general, finding the completeness threshold is as hard as actually verifying the system, but many errors are shallow, and may be found with a low value of k. Therefore, using even a fairly small value can increase one's confidence in a system, even if one can still not be *completely sure* that the system is correct. Since this approach involes generating a formula for a SAT-solver, the efficiency of any BMC tool is tied to the performance of current SAT-solvers. SAT-solvers have become increasingly efficient over recent years (see Figure 1), and BMC has therefore become more and more effective for model verification.



Credit: Daniel Le Berre

Figure 1: The increasing performance of modern SAT-solvers.

For a given proposition about a system, there are multiple ways of encoding that proposition as a propositional formula, and the encoding we choose can affect the performance of a SAT-solver on that formula. For example, the naive method for encoding "at-most-one" constrains (that is, constraints of the form $x_1 + x_2 + \cdots + x_n \leq k$ where $x_i \in \{0, 1\}$) produces large formulas that lead to bad performance for SAT-solvers. A more intelligent approach to encoding leads to much better performance (See Section 2.2 for more details).

Given that choice of encoding can have a significant effect on both the complexity of the formula produced, and on the subsequent performance of a SAT-solver on that formula, it is natural to attempt to preprocess a formula before running a SAT-solver, an an attempt to shrink the formula or otherwise mould it more to the specific abilities of the solver. A number of attempts have been made at this, some quite successful. The best approach to preprocessing discovered so far relies on variable elimination, and both significantly decreases the size of many formulas, and also improves the

performance of SAT-solvers on such formulas [Eén and Biere, 2005].

One possible approach to preprocessing is the application of *extended resolution*. Runs of SAT-solvers can be thought of as working within the *resolution* proof system, which has one rule (introduced in detail in Section 3). Extended resolution adds an additional rule, allowing the introduction of a fresh varaible $x \leftrightarrow a \vee b$, where a and b are literals already introduced. Extended resolution facilitates proofs of some theorems (for example, the pigeonhole principle) in a polynomial number of steps where resolution requires an exponential number of steps. Motivated by this improvement, I investigated — through the use of genetic algorithms, a technique from artificial intelligence — whether extended resolution can be employed in a preprocessing step in such a way as to improve SAT-solver performance on formulas obtained from BMC on software verification problems.

This report will cover:

- Encodings and preprocessing in the context of CDCL SAT-solvers.
- Resolution and extended resolution, exploring in detail how extension allows the pigeonhole principle to be proved in a polynomial number of steps.
- A discussion of strategies already employed for using extended resolution in SAT-solving, both in a preprocessing step and inside a SAT-solver itself.
- Genetic algorithms and how they can be applied to the problem of finding good sets of extensions to perform during preprocessing.
- An implementation of the technical infrastructure needed to apply genetic algorithms to this problem.
- Anlysis of the results of applying genetic algorithms to the problem.
- Discussion of what the results mean for the search for a good way to incorporate extended resolution into the general process for SAT-solving.

2 Boolean satisfiability

A *propositional formula*, or *Boolean expression*, consists of one of: a Boolean variable (a variable that has the value 'true' or 'false'), the negation of a

Boolean variable (written "$\neg x$" if x is a Boolean variable), the conjunction of two propositional formulas (written "$\phi \wedge \psi$" if ϕ and ψ are propositional formulas), and the disjunction of two propositional formulas (written "$\phi \vee \psi$" if ϕ and ψ are propositional formulas). A Boolean variable or its negation is called a *literal*.

Given a propositional formula, and an assignment of Boolean values to every variable found in the formula, the formula may be *evaluated* by replacing each literal with the corresponding value from the assignment, and then replacing $\phi \wedge \psi$ with 'true' iff ϕ and ψ both evaluate to true, and replacing $\phi \vee \psi$ with 'true' iff either ϕ or ψ evaluates to 'true'. Otherwise these operators are replaced with 'false'.

The *Boolean satisfiability problem* asks whether, given a particular propositional formula, there exists a *satisfying assignment* to the variables in the formula. That is, does there exist an assignment under which the formula evaluates to 'true'? If this is the case, then the formula is said to be *satisfiable*, or "SAT". Otherwise, the formula is said to be *unsatisfiable* ("UNSAT"). A computer program that solves the Boolean satisfiability problem for a given formula is called a SAT-solver. A great deal of reasearch has been carried out into the creation of effective SAT-solvers.

Modern SAT-solvers operate on a machine-readable representation of formulas in *conjunctive normal form* (CNF). For every propositional formula, there is an equivalent CNF formula, which can be found by repeated application of DeMorgan's laws, and the laws about distributivity and double-negatives [Plaisted and Greenbaum, 1986]. Therefore SAT-solvers can effectively operate on all propositional formulas.

A CNF formula is defined as a conjunction of clauses, where each clause is a disjunction of literals:

$$(l_1 \vee l_2 \vee \cdots \vee l_j) \wedge (l_{j+1} \vee l_{j+2} \vee \cdots \vee l_k) \wedge \cdots \wedge (l_{m+1} \vee l_{m+2} \vee \cdots \vee l_n)$$

Note that this formula just requires one literal from each clause to be true.

2.1 CDCL SAT-solvers

The most efficient modern SAT-solvers operate according to versions of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. The basis of this algorithm is essentially a depth-first search with backtracking of the whole

space of partial assignments, looking for a full assignment that satisfies the formula. In the worst case, this type of exhaustive search would take $O(2^n)$ time to complete for a formula of n variables.

SAT-solvers can achieve much better performance in practice through the use of a number of heuristics. The DPLL algorithm introduced *unit propagation*, which is the immediate assignment of the required value to any variables found on their own in a clause, and *pure literal elimination*, which is the immediate assignment to any variable which is only found in one polarity in the whole formula (thereby making every clause containing that variable true) [Davis et al., 1962].

The latest SAT-solvers, such as MiniSat, heavily exploit further optimisations such as faster data structures for the backtracking algorithm, better strategies for picking which variable to set at each step, and more advanced strategies for backtracking when a conflict is found [Eén and Sörensson, 2004]. In particular, *conflict driven clause learning* (CDCL) aims to infer a clause from a conflict which encapsulates, in some sense, the 'reason' for the conflict, and causes the backtracking algorithm to fail more quickly on other paths that would lead to failure for the same reason [Marques-Silva and Sakallah, 1999, Moskewicz et al., 2001].

2.2 Encodings and preprocessing

The choice of encoding can dramatically affect the size of a formula representing some proposition about a system. Let $x_1, x_2, \ldots, x_n \in \{0, 1\}$, and consider propositions of the form

$$x_1 + x_2 + \cdots + x_n \leq k.$$

A naive encoding of this constraint requires $\binom{n}{k+1}$ clauses. For example, let $x_1, \ldots, x_4$ be Boolean variables, and require that

$$x_1 + x_2 + x_3 + x_4 \leq 1.$$

For a naive encoding of this proposition, it is sufficient that each pair of variables include one that is false, since that will entail that at most one variable is true in the whole set, which satisfies the requirement above. This encoding produces

$$\begin{aligned}(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \\ \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4).\end{aligned}$$

Even in this case, where $k = 1$, $O(n^2)$ clauses are required. There are, however, a number of approaches to encoding these constraints that lead to much smaller formulas [Biere et al., 2014]. For example, the formula above can be encoded by first arranging the four variables in a matrix:

	1	2
1	x_1	x_2
2	x_3	x_4

Then we introduce a variable c_n for each column n, and a variable r_m for each row m. We then add two clauses for each variable x_i. If x_i is in column n and row m, we add

$$x_i \to c_n = \neg x_i \vee c_n,$$
$$x_i \to r_m = \neg x_i \vee r_m.$$

In the above case, we add $(\neg x_3 \vee c_1) \wedge (\neg x_3 \vee r_2)$ for the x_3 variable, for example.

These clauses mean that any of the variables in column n being true will cause c_n to be true, and any of the variables in row m being true will cause r_m to be true. Then, we simply require that at most one column, and at most one row contains a true variable, so

$$(\neg c_1 \vee \neg c_2) \wedge (\neg r_1 \vee \neg r_2).$$

It is possible to arrange n variables in a p by q matrix where $p = \lfloor \sqrt{n} \rfloor = O(\sqrt{n})$ and $q = \lceil n/p \rceil = O(\sqrt{n})$. Therefore, to encode the formula with this approach, we need $O(n)$ clauses specifying the location of each variable, $O(p^2) = O(n)$ clauses ensuring that only one row contains a true variable, and $O(q^2) = O(n)$ clauses ensuring that only one column contains a true variable. Therefore this encoding requires only $O(n)$ clauses in total, a huge (asymptotic[1]) improvement over the $O(n^2)$ clauses required by the naive encoding.

Different encodings lead not just to different sizes of formula, but often radically different performances by SAT-sovlers [Martins et al., 2011]. This suggests the idea of pre-processing a formula before the SAT-solver is run. It may be possible to detect opportunities to re-structure a formula in such a way that the SAT-solver returns the result in less time — essentially to change the encoding after the fact. If the time saved by the SAT-solver is

[1]The encoding is better in absolute terms when $n > 47$ [Biere et al., 2014].

greater than the time taken to detect and apply the optimisations, then a net gain is made.

Many preprocessing techniques simply attempt to decrease the overall size of the formula, in a similar way to the way the encoding of at-most-one problems above reduced the size of the formula. Others manipulate the formula in other ways, exploiting different types of structure in the formula. As mentioned in the introduction, good results have been obtained through a preprocessor that performs variable elimination [Eén and Biere, 2005]. An implementation of several techniques in am odarl way within in one preprocessor, "Coprocessor 2.0", most operating on the idea of eliminating some kind of repeated structure in the formula, also led to improvements on various instances, and the modular nature of the preprocessor means that techniques that do not work well on particular types of formula can be turned off for those formulas [Manthey, 2012]. A preprocessing technique based on extended resolution does not aim to reduce the size of the formula, but aims to exploit similarity found between clauses in the formula to eliminate sequences of steps taken by the SAT-solver that apply to different — but similar — clauses, but could be performed in a single sequence of steps after the extension.

3 Resolution and extended resolution

The *pigeonhole principle* states that, if $n > m$, it is impossible to place n items in m bins without any bins containing more than one item. We can attempt to verify this by encoding the contrapositive, "it is possible to place $n > m$ items in m bins without any bin containing more than one item" as a propositional formula, and then showing that it is UNSAT. As n could be arbitrarily larger than m, we will actually show that the propositional formula encoding the assertion "it is possible to place n items in $n-1$ bins without any bin containing more than one item" is UNSAT. If it is impossible to place n items in $n-1$ bins in this way, then it is certainly impossible to place even more than n items (since we have to place n items in $n-1$ bins on the way), so this assertion suffices.

Let $P_{i,b}$ be a propositional variable standing for item i being placed in bin b. Then we start by asserting that, for all items $1 \leq i \leq n$,

$$P_{i,1} \vee P_{i,2} \vee \cdots \vee P_{i,n-1}.$$

That is, for every item, it's either in bin 1, bin 2, ..., or bin $n-1$ — so every item is in a bin. Then we assert that, for every bin $1 \leq b \leq n-1$, for each pair of items $1 \leq i < j \leq n$,

$$\neg P_{i,b} \vee \neg P_{j,b}.$$

That is, for each bin, for each pair of items, one or other of those items is not in the bin — so no pair of items is in the same bin. Together, these clauses assert that it is possible to place n items in $n-1$ bins without more than one item in any bin. This formula is UNSAT.

One way of proving that this formula is UNSAT is to provide a resolution refutation. Where clauses are respresented as sets of literals (e.g. $a \vee \neg b$ is represented by $\{a, \neg b\}$), the resolution rule works as follows.

$$\frac{C_1 \cup \{L\} \quad C_2 \cup \{\neg L\}}{C_1 \cup C_2} \text{ res}$$

For example, we can use resolution to prove the pigeonhole principle for the easy $n = 2$ case (where there is only one bin):

$$\dfrac{\dfrac{\{P_{1,1}\} \quad \{\neg P_{1,1}, \neg P_{2,1}\}}{\{\neg P_{2,1}\}} \text{ res} \qquad \{P_{2,1}\}}{\bot} \text{ res}$$

This is, in fact, a *tree resolution* proof, because no clause is an antecedent of more than one other clause. Unfortunately, for sufficiently large n, a tree resolution proof of the pigeonhole principle for n items requires $2^{\Omega(n)}$ steps, suggesting that tree resolution may not be the best proof system with which to prove the pigeonhole principle [Haken, 1985]. However, runs of DPLL-based SAT-solvers on UNSAT formulas correspond to tree resolution refutations of the formula, which suggests that such SAT-solvers will exhibit low performance on instances of the pigeonhole principle, and perhaps other similar formulas [Rossi et al., 2006]. This raises the question of whether there is a more powerful system that allows a sub-exponential length proof – and thus better performance by a DPLL-based SAT-solver – on this type of instance. These systems do exist, and one such system is *extended resolution*.

Extended resolution adds an additional rule beyond the resolution rule. Given a pair of literals l_1 and l_2 from the propositional formula, and a fresh

variable x, we may introduce a new set of clauses representing $x \leftrightarrow l_1 \vee l_2$ [Tseitin, 1983]. The set of clauses representing this addition is

$$\begin{aligned} x \leftrightarrow l_1 \vee l_2 &= (x \rightarrow (l_1 \vee l_2)) \wedge (l_1 \rightarrow x) \wedge (l_2 \rightarrow x) \\ &= (\neg x \vee l_1 \vee l_2) \wedge (\neg l_1 \vee x) \wedge (\neg l_2 \vee x). \end{aligned}$$

Extended resolution permits a polynomial-length proof of the pigeonhole principle [Cook, 1976], which is outlined below.

A conventional proof of the pigeonhole principle works by induction on n. First, it can easily be shown that it's not possible to assign 2 items to 1 bin without the sole bin containing more than one item (it will have to contain both!), as we did by resolution above. We will see this as the lack of an injective mapping from $\{1, 2\}$ to $\{1\}$. Then, for the inductive step, we show that if there is a mapping from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n-1\}$, then there is a mapping from $\{1, 2, \ldots, n-1\}$ to $\{1, 2, \ldots, n-2\}$. We do this by supposing that ϕ is an injective mapping from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n-1\}$, in which case

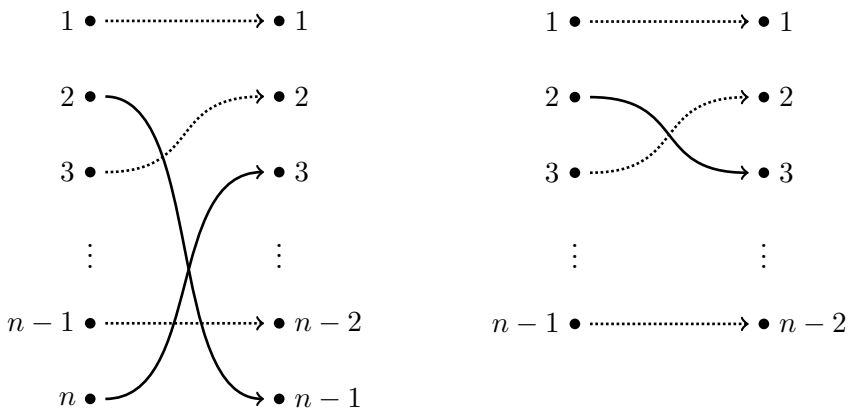
$$\phi'(i) = \begin{cases} \phi(n) & \text{if } \phi(i) = n-1, \\ \phi(i) & \text{otherwise} \end{cases}$$

is an injective mapping from $\{1, 2, \ldots, n-1\}$ to $\{1, 2, \ldots, n-2\}$. Intuitively, ϕ' maps everything in the same way as ϕ, except that the mappings *from* n and *to* $n-1$ are merged, allowing those numbers to be deleted (see Figure 2). Since there isn't a mapping for $n = 2$, by contradiction there can't be one for $n = 3$, and therefore there is none for $n = 4$, and so on.

The extended resolution proof mirrors this process. Suppose we have the clauses restricting the values of $P_{i,b}$ for some instance of the pigeonhole principle, as given above. These clauses, if they are satisfiable, imply some mapping ϕ. We now introduce new propositional variables $Q_{i,b}$, with $1 \leq i \leq n-1$ and $1 \leq b \leq n-2$, which, if satisfiable, describe the corresponding mapping ϕ'. We set

$$\begin{aligned} Q_{i,b} &= P_{i,b} \vee (P_{i,n-1} \wedge P_{n,b}) \\ &= (P_{i,b} \vee P_{i,n-1}) \wedge (P_{i,b} \vee P_{n,b}), \end{aligned}$$

which just says that the new mapping of a particular item ($Q_{i,b}$) either matches the old one ($P_{i,b}$) or it represents that merge between the mappings



(a) An example mapping ϕ (b) The derived ϕ'

Figure 2: An illustration of the inductive step in proving the pigeonhole principle.

from n and to $n-1$ ($P_{i,n-1} \wedge P_{n,b}$). This can be achieved by performing two extensions per variable $Q_{i,b}$:

$$Q_{i,b} \leftrightarrow P_{i,b} \vee P_{i,n-1}$$
$$Q_{i,b} \leftrightarrow P_{i,b} \vee P_{n,b}$$

From this new (polynomial-size) set of clauses, and the clauses given by the instance of the pigeonhole principle for n items, we can obtain, by a polynomial number of resolutions, the following clauses:

$$\begin{array}{lll} & Q_{i,1} \vee Q_{i,2} \vee \cdots \vee Q_{i,n-2} & \text{for } 1 \leq i \leq n-1, \\ \text{and} & \neg Q_{i,b} \vee \neg Q_{j,b} & \text{for } 1 \leq b \leq n-2 \text{ and } 1 \leq i < j \leq n-1. \end{array}$$

(See Appendix A for a worked example.) These clauses describe the pigeonhole principle for $n-1$ items. Repeating this process of extension and resolution $O(n)$ times, until we reach the $n = 2$ case, gives a polynomial-length proof of the pigeonhole principle.

4 Strategies for extension

We have seen by the example of the pigeonhole principle that adding extension to the resolution proof system can sometimes permit polynomial-length proofs of assertions that would otherwise require exponential length. Since tree resolution proofs correspond to runs of DPLL-based SAT-solvers, adding the ability to perform extension in the process of SAT-solving can potentially improve the performance of SAT-solvers.

In 2010, a restriction of extended resolution was demonstrated which may be used to create a SAT-solver that, in some cases, outperforms ones that do not use extension [Audemard et al., 2010]. The authors note that, prior to their paper, no SAT-solver based on extended resolution had been proposed, because "it is hard to devise a heuristic that introduces variables that are likely to help find a short refutation". In other words, it is unclear which extensions to perform, in order to increase performance. They propose restricting the situations where extension may be performed to those where there are two clauses, $C_1 = l_1 \vee \alpha$ and $C_2 = l_2 \vee \beta$, where neither α nor β contain negations of any literals contained in the other, in which case extension may be performed to introduce $z \leftrightarrow \neg l_1 \vee \neg l_2$. The aim of doing so is to avoid redundancy in any later sequence of resolutions producing a new pair of clauses which essentially only differ in that one includes l_1 and the other l_2. They describe a performant way of implementing this strategy that outperformed state-of-the-art SAT-solvers on instances from recent competitions.

In 2015, a group of students explored the possibility of using *common subexpression elimination* (CSE) in various ways to improve the performance of SAT-solving [Yan et al., 2015]. They suggest eliminating certain repeated subexpressions by introducing new variables using a generalised version of extension (although they do not actually mention extended resolution themselves). If $\{l_1, l_2, \ldots, l_n\}$ is a set of literals from a propositional formula and x is a fresh variable, then we can introduce the clauses necessary to establish

$$x \leftrightarrow l_1 \vee l_2 \vee \cdots \vee l_n$$

and then, in any clause containing all of $l_1, l_2, \ldots, l_n$, delete those literals, and add x. Three methods are given for choosing which subexpressions to eliminate. In the first, the common subexpressions are chosen such that, after they are eliminated, the problem is of minimal size. The authors conjecture that this problem is NP-hard, and give two approximate algorithms. One

is based on replacing subexpressions that are found with frequency above a given threshold, and the other is based on the LZW string-compression algorithm. All of these strategies are implemented as a pre-processing step before running the SAT-solver. This approach is attractive because of its simplicity, and the authors find an improvement in performance in about half the cases they try.

Formulas obtained from BMC in software verification may have many sets of literals that are found in many clauses. Extended resolution could be used to replace those sets of literals with a single variable, which may improve the performance of a SAT-solver on the formula, as in the findings of Yan et al. However, it is difficult to know which sets of replacements to perform. Replacing every repeated subclause will not necessarily improve performance, but there may be sets of replacements that do. This raises the question of whether there is usually such a set of replacements on these formulas. For this project, I attempted to answer the question:

> Are there sets of subclause replacements that improve the performance of the MiniSat SAT-solver on certain CNF formulas obtained from BMC in software verification problems, and if so, can these be found by applying a genetic algorithm?

5 Genetic algorithms

A *genetic algorithm* is a search algorithm that mimics natural selection [Goldberg, 1989]. In the most basic form of the algorithm, sufficiently long binary strings are used to represent points in the search space, and then an initial population of random strings is created. Each of the strings is scored according to the fitness of the corresponding item in the search space, and then a sample of the most highly-scored strings is taken. From this sample, a new population the same size as the original is created, by taking random pairs of strings from the sample and combining them in a specific way (see below). This process of scoring, sampling, and combining is repeated a number of times, in the hope of finding a member of the search space with high fitness.

In the step where a new population is being generated from the high-fitness sample, the new population members are usually generated according to the following algorithm:

1. **Selection:** Randomly choose two *parents* a and b from the high-fitness sample.
2. **Crossover:** Where n is the length of the strings, pick some integer k such that $0 \leq k \leq n$, and then combine the two parent strings to create the new member $m = a[0 \ldots k) \parallel b[k \ldots n)$ (See Figure 3).
3. **Mutation:** Randomly flip some of the bits in m and then add it to the new population.

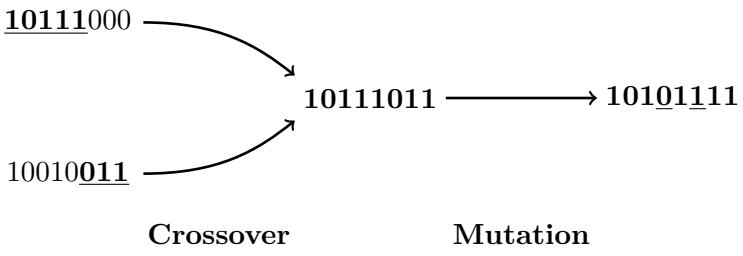


Figure 3: An illustration of the generation of a new member from a pair of parents in a genetic algorithm.

The aim of the crossover part of the algorithm is to allow good features that are developed in separate members of the population to be combined as "building blocks", sometimes leading to a member of the population that is stronger than either of the original members [Goldberg, 1989]. In some variations on the basic genetic algorithm, multiple "cuts" are made so that more than two members of the population may be combined, but this hasn't been found to produce better final outcomes in most cases, and can lead to premature convergence, owing to a loss of genetic diversity in the population [Esquivel et al., 1997].

The aim of the mutation part of the algorithm is to allow the exploration of "nearby" members of the population. If the search space is sufficiently smooth with respect to the fitness function, then this leads to a hillclimbing effect. This also means that care must be taken with the encoding of solutions as bit vectors, and when setting the rate of mutation, since certain encodings or rates of mutation may not lead to a sufficiently similar "neighbouring" solutions for the hillclimbing effect to work correctly [Taherdangkoo et al., 2013]. In general, the probability of any bit in a member of the population being flipped should not be above 0.05, and with a mutation rate below that threshold, the population size should be between approximately 30 and 100 [Grefenstette, 1986].

There are several different strategies for the selection phase, where the high-fitness sample is chosen [Thierens and Goldberg, 1994]. The simplest is *truncation selection*, which is the outcome of sorting the population by fitness, and taking the best m members, for some m. An alternative approach is *fitness proportionate selection*, also called *roulette-wheel selection*, where we calculate the fitness of each individual, and then assign each individual some part of the $[0, 1]$ interval, so that the size of each individual's section of the interval is proportional to its fitness. Then a random point in the $[0, 1]$ interval is picked, and the corresponding member of the population is added to the high-fitness sample. This is repeated t times, where t is the desired size of the high-fitness sample. Both of these strategies generally perform well, but fitness proportionate selection can run in to problems at later stages of execution when the fitness values of the population are similar, as it might not provide enough selection pressure to improve the population [Whitley et al., 1989]. As the fitness values of members of the population in this project are indeed likely to be similar, I chose to use truncation selection. Often, the strong members from the previous iteration are retained in the population for the new iteration without going through crossover and mutution (which are then used to generate the remainder of the new population). This approach is called *elitism*, and has the advantage of ensuring that strong solutions are not lost in the mutation step.

In this project, based on my findings above, I used a population size of 50 with elitism, and a mutation probability of 0.01. Each member of the population describes which of the possible subclause replacements should be made for a formula. The fitness of a member of the population is the time taken by MiniSat to solve the formula after the replacements have been made (so smaller numbers indicate a higher fitness). The genetic algorithm will then attempt to find the set of subclause replacements that leads to the best overall performance.

6 Technical implementation

In the technical implementation of this project, I obtained a number of benchmarks that come from real-world bounded model checking. The benchmarks come from implementations of the Newton-Raphson method, polynomial approximations of sine, and squaring of floating-point numbers. CBMC was used to generate the formulas from C programs (See Appendix **??** for examples). The programs do not contain any loops, which means that a

cut-off point for unrolling the loops does not need to be chosen.

To execute the genetic algorithm on a given benchmark, we follow this algorithm:

1. Find, and number, all of the repeated subclauses of size 2 or 3.
2. Generate an initial population of 50 bit vectors, each having one bit per subclause.
3. For each member of the population, create a new benchmark file by performing the required extensions.
4. Run MiniSat on each member of the population, recording the time taken.
5. If the average fitness for the whole population has converged, terminate.
6. Rank members of the population by their MiniSat time, and take the best half.
7. Generate a new population of 50 comprising the best half of the previous population, along with 25 created from that half by crossover and mutation.
8. Go to Step 3.

Finding all of the repeated subclauses of size 2 or 3 could be done in many ways. For example, simply passing over the file counting the number of occurrences of each subclause would work, particularly when each clause is small. This problem has been studied relatively extensively within the field of database mining, where fast algorithms such as the *Apriori algorithm* have been discovered, which scale up effectively to very large datasets [Agrawal et al., 1994]. A detailed investigation of these algorithms is beyond the scope of this project, since the goal is simply to determine whether there are sets of subclause replacements which improve SAT-solver perfomance. For Step 1 of the above algorithm, I simply call an existing program that uses the Apriori algorithm [Borgelt, 2003].

I represent a propositional formula by an instance of the `Formula` class (See Figure 4). The constructor takes as an argument a CNF formula in the standard DIMACS format, and stores it internally as a list of lists of integers. Each inner list represents a clause, and each integer represents a literal. The

```
class Formula:
    def __init__(self, raw_cnf):
        self.next_fresh = 0
        self.literal_locations = {}
        self.clauses = []
        lines = raw_cnf.split("\n")
        for line in lines:
            if len(line) == 0 or line[0] == "c" or line[0] == "p":
                pass
            else:
                parts = [int(part) for part in line.strip().split("␣") if part != ""]
                self.add_clause(parts[:-1])

    def add_clause(self, literals):
        self.clauses += [literals]
        # Do some extra processing to help the find_clauses_containing method
        ...

    def find_clauses_containing(self, literals):
        # Quickly find all of the clauses containing the given literals
        ...

    def extend(self, variables):
        x = self.get_fresh_variable()
        var_set = set(variables)
        for clause_number in self.find_clauses_containing(variables):
            new_clause = [x]
            for literal in self.clauses[clause_number]:
                if literal not in var_set:
                    new_clause += [literal]
            self.clauses[clause_number] = new_clause
        self.add_clause([-x] + variables)
        for variable in variables:
            self.add_clause([-variable, x])

    def to_cnf_file(self):
        result = "p␣cnf␣" + str(self.next_fresh) + "␣" + str(len(self.clauses))
        for clause in self.clauses:
            result += "\n" + "␣".join(str(l) for l in clause) + "␣0"
        return result
```

Figure 4: The essential methods of the `Formula` class. (Some helper code, which allows the class to be more efficient, has been omitted in the interest of clarity.)

integer x represents the literal l_x and the integer $-x$ represents the literal $\neg l_x$.

The `Formula` class includes a method `extend` which takes a list of literals $[l_1, l_2, \ldots, l_n]$ and adds a fresh variable x such that

$$x \leftrightarrow l_1 \vee l_2 \vee \cdots \vee l_n.$$

It also finds every clause containing all of $l_1, l_2, \ldots, l_n$, and replaces those literals with a single reference to x. This has the effect of replacing that subclause with a single variable representing it, across the whole formula. The class also has a method `to_cnf_file` which can produces a string containing the current state of the formula in the DIMACS format.

For the genetic algorithm, the population is handled by the `Population` class (see Figure 5). The initial population is generated in the `__init__` method (the constructor) by creating 48 (or 2 less than `population_size` if that is not 50) random bit vectors of the appropriate length. We also add the vectors $[111\cdots1]$ and $[000\cdots0]$, in case either of these extremes turns out to perform well. The bit vectors are represented as lists of Boolean values. This is not very efficient in memory, but it doesn't matter because the population is quite small.

The `Population` class also handles the logic of each iteration of the genetic algorithm. The `get_best_few` method takes as an argument a function `scoring_function` and applies this funciton to each member of the population. The `scoring_function` returns two values: the fitness of the population member (where a smaller value implies a fitter member), followed by an object possibly containing other data gathered during the scoring (which may be of interest when it comes to analysis of the run).

The run is set up by three functions that make use of the `Formula` and `Population` classes (See Figure 6). The function `run_solver` takes a `Formula` and a list of subclauses (so a list of lists of literals), performs the necessary extensions to replace the subclauses with representative literals, and writes the new formula to a temporary file. It then makes a system call to execute MiniSat, and gets the output as a string. That string is fed into the `SolverResult` helper class, which simply extracts the interesting information from the MiniSat output and stores it for easy access later.

The `run_solver` function is used by the `run_genetic_algorithm` function. This function takes as arguments the formula in question, along with a list of all of the possible subclauses that could be replaced (as found by

```
class Population:
    def __init__(self, string_length, size=50, select_best=25):
        self.size = size
        self.select_best = select_best
        self.members = [[random_bit() for j in xrange(string_length)]
                        for i in xrange(size - 2)]
        self.members += [[True for i in xrange(string_length)]]
        self.members += [[False for i in xrange(string_length)]]

    def get_best_few(self, scoring_function):
        scores = []
        for member in self.members:
            this_score, other_data = scoring_function(member)
            scores += [{
                "member": member,
                "score": this_score,
                "other_data": other_data
            }]
            scores.sort(key=lambda s: s["score"])  # smaller is better
        total_fitness = sum([score["score"] for score in scores])
        return (total_fitness, [i["member"] for i in scores[:self.select_best]], scores)

    def generate_new_members(self, best_few, mutation_probability=0.01):
        self.members = best_few
        for member_count in xrange(len(best_few), self.size):
            left_parent = random.choice(best_few)
            right_parent = random.choice(best_few)
            # Crossover
            assert len(left_parent) == len(right_parent)
            crossover_point = random.randint(0, len(left_parent))
            new_member = left_parent[:crossover_point] \
                + right_parent[crossover_point:]
            # Mutation
            for i in xrange(len(new_member)):
                if random.random() < mutation_probability:
                    new_member[i] = not new_member[i]
            self.members += [new_member]

    def improve(self, scoring_function, mutation_probability):
        total_fitness, best_few, all_scores = self.get_best_few(scoring_function)
        self.generate_new_members(best_few, mutation_probability)
        return total_fitness
```

Figure 5: The essential methods of the `Population` class. (Code that performs logging, and other secondary functions like converting the population to and from strings, is not included here, as it is immaterial to the behaviour of the genetic algorithm.)

```
def run_solver(cnf, replacements):
    new_formula = cnf.clone()
    for replacement in replacements:
        new_formula.extend(replacement)
    with open("replaced.cnf", "w") as cnf:
        cnf.write(new_formula.to_cnf_file())
    output = os.popen("./minisat_replaced.cnf").read()
    os.system("rm_replaced.cnf")
    return SolverResult(output)

def get_subclauses_from_mask(subclauses, mask):
    used_subclauses = []
    for i in xrange(len(mask)):
        if mask[i]:
            used_subclauses += [subclauses[i]]
    return used_subclauses

def run_genetic_algorithm(formula, subclauses, threshold=0.005):
    population = genetic.Population(string_length=len(subclauses))
    def scoring_function(mask):
        used_subclauses = get_subclauses_from_mask(subclauses, mask)
        result = run_solver(formula, used_subclauses)
        return result.time, result.get_bundle()
    good_example = population.get_good_example(scoring_function,
                                               threshold=threshold)
    return get_subclauses_from_mask(subclauses, good_example)
```

Figure 6: The functions used to run the genetic algorithm on the subclause sets. (Code that performs logging has been omitted.)

the Apriori program discussed above), and generates an instance of the `Population` class. The length of the strings in the population is the same as the number of subclauses that could be replaced, and each string is essentially a bitmask describing which of the subclauses should actually be replaced. Then it creates a closure `scoring_function` which takes such a mask as an argument and obtains the actual list of subclauses to be replaced using the `get_sublcauses_from_mask` function, before running the solver with those replacements, and getting the time taken by MiniSat. This closure is passed as the `scoring_function` argument to `population.get_good_example`, which is a method in the `Population` class which simply calls the `improve` method repeatedly until the average fitness of the population hasn't changed more by more than 0.5% between each of three consecutive iterations, and then returns the fittest member of the final pop-

ulation.

7 Analysis of results

I ran the genetic algorithm on 19 instances from ???. The overall results can be found in Table 1. Like those of Yan et al., my results show that performing all possible replacements leads to an improvement in some cases, and not in others. The genetic algorithm, however, was able, in all cases, to find sets of replacements that lead to an improvement in performance, and often the improvement was significant. On average, performing the best set of replacements found by the algorithm lead to an improvement in performance of 85% compared with making no replacements. Therefore, if we had an oracle that quickly produced such sets of replacements from formulas, we would have a way to dramatically improve the performance of SAT-solvers on benchmarks from (at least) this domain.

7.1 Effectiveness of the genetic algorithm

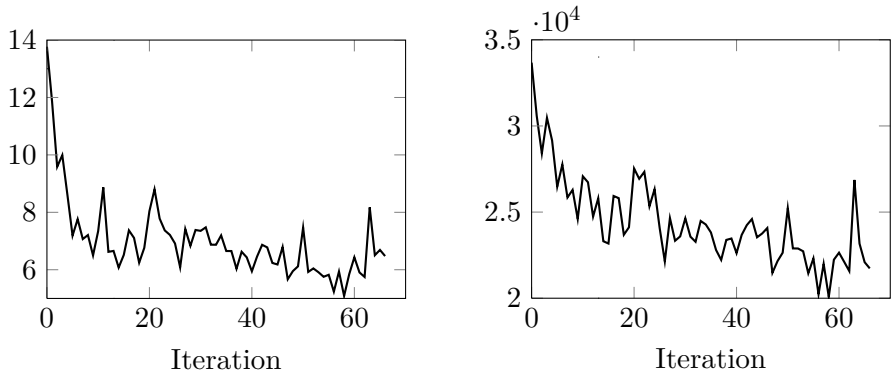
The average time taken by MiniSat for each member of the population roughly agrees with the average number of conflicts found by MiniSat on the same runs (see Figure 7 for an example). This suggests that the time is not being significantly influenced by the platform on which the genetic algorithm is being run, since ???.

Because runs of SAT-solvers on UNSAT formulas correspond to tree-resolution refutations, improvements on UNSAT formulas are particularly interesting. Such improvements must rely on an optimisation in the proof arising from some change performed during the pre-processing step, whereas runs on SAT formulas could be improved by the pre-processing merely happening to cause a built-in 'magic' heuristic to work better in that particular case. Therefore it makes sense to ask how the performance of the genetic algorithm on SAT benchmarks differs from the performance on UNSAT benchmarks.

The genetic algorithm does find good solutions on both SAT and UNSAT benchmarks. However, it tends to converge on one strong solution more often in SAT instances than it does in UNSAT instances. To see this, we need to investigate how the diversity of the population varies over the iterations during the run.

Benchmark	SAT?	Clauses	Variables	Replacements		MiniSat Time (seconds)		
				Possible	Best	No reps.	All reps.	Best reps.
sine_2_false–unreach–call	SAT	26,219	121,870	2192	1156	3.11	1.20	0.10
square_2_false–unreach–call	SAT	26,266	121,596	2192	1079	2.71	1.30	0.08
square_1_false–unreach–call	SAT	26,267	121,599	2192	1305	4.02	1.90	0.07
sine_1_false–unreach–call	SAT	26,513	123,002	2192	1113	2.27	1.79	0.05
sine_3_false–unreach–call	SAT	26,241	121,936	2192	1093	3.61	0.30	0.07
square_3_false–unreach–call	SAT	26,267	12,1599	2192	1090	4.64	3.87	0.05
newton_1_8_false–unreach–call	SAT	54,744	253,226	5044	2469	6.60	4.72	0.31
newton_1_7_false–unreach–call	SAT	54,772	253,310	5044	2491	12.89	4.93	1.58
newton_1_5_false–unreach–call	SAT	54,751	253,247	5044	2477	12.31	9.38	0.57
newton_1_6_false–unreach–call	SAT	54,773	253,313	5044	2555	15.33	17.31	0.49
newton_1_4_false–unreach–call	SAT	54,773	253,313	5044	2529	23.07	14.08	1.26
newton_2_6_false–unreach–call	SAT	109,084	504,912	7236	3623	44.05	92.81	11.12
newton_2_8_false–unreach–call	SAT	109,055	504,825	7236	3556	129.15	108.25	5.14
sine_8_true–unreach–call	UNSAT	26,212	121,849	2192	1072	5.30	13.72	1.36
sine_7_true–unreach–call	UNSAT	26,220	121,873	2192	1110	6.41	26.42	1.62
newton_1_1_true–unreach–call	UNSAT	54,773	253,313	5044	2480	39.82	209.66	22.77
newton_1_2_true–unreach–call	UNSAT	54,773	253,313	5044	2487	62.23	123.53	26.80
sine_6_true–unreach–call	UNSAT	26,241	121,936	2192	1139	71.04	542.11	1.98
square_7_true–unreach–call	UNSAT	26,267	121,599	2192	1090	131.38	76.61	9.16

Table 1: Benchmark results



(a) Mean MiniSat time (s)

(b) Mean conflict count

Figure 7: **newton_1_8_false–unreach–call**: Average time and average number of conflicts correlate.

In order to see how the diversity of the population varied over the run of the genetic algorithm, I computed, for each iteration, the *Shannon entropy* of the action of selecting a member from the population [Shannon, 1948]. That is, given the whole population, how many bits of information do we expect to gain by selecting one member of the population? If this is equal to the length of the vectors in the population, then each bit is 1 in as many members of the population as it is 0, so we might say that the population is very diverse. If it is 0, then all members of the population are the same. In the initial population, we expect the Shannon entropy to be almost maximised, and it should decrease over time, eventually becoming very small if the algorithm converges on one solution. We compute the Shannon entropy by summing the binary entropy function H_b over each bit position in the population. Precisely, let p_i be the fraction of members of the population in which the bit at location i is a 1, and N be the length of the population members. Then the Shannon entropy is

$$H(\text{pop}) = \sum_{i=0}^{N-1} H_b(p_i) = \sum_{i=0}^{N-1} \left(-p_i \log_2(p_i) - (1-p_i)\log_2(1-p_i)\right).$$

Figure 8 shows a typical run of the genetic algorithm on a SAT benchmark. The range of the number of replacements being made decreases steadily over time, closing in on one number of replacements. The population diversity — as measured by the Shannon entropy — also decreases over time until the algorithm successfully converges on one small group of actual solutions

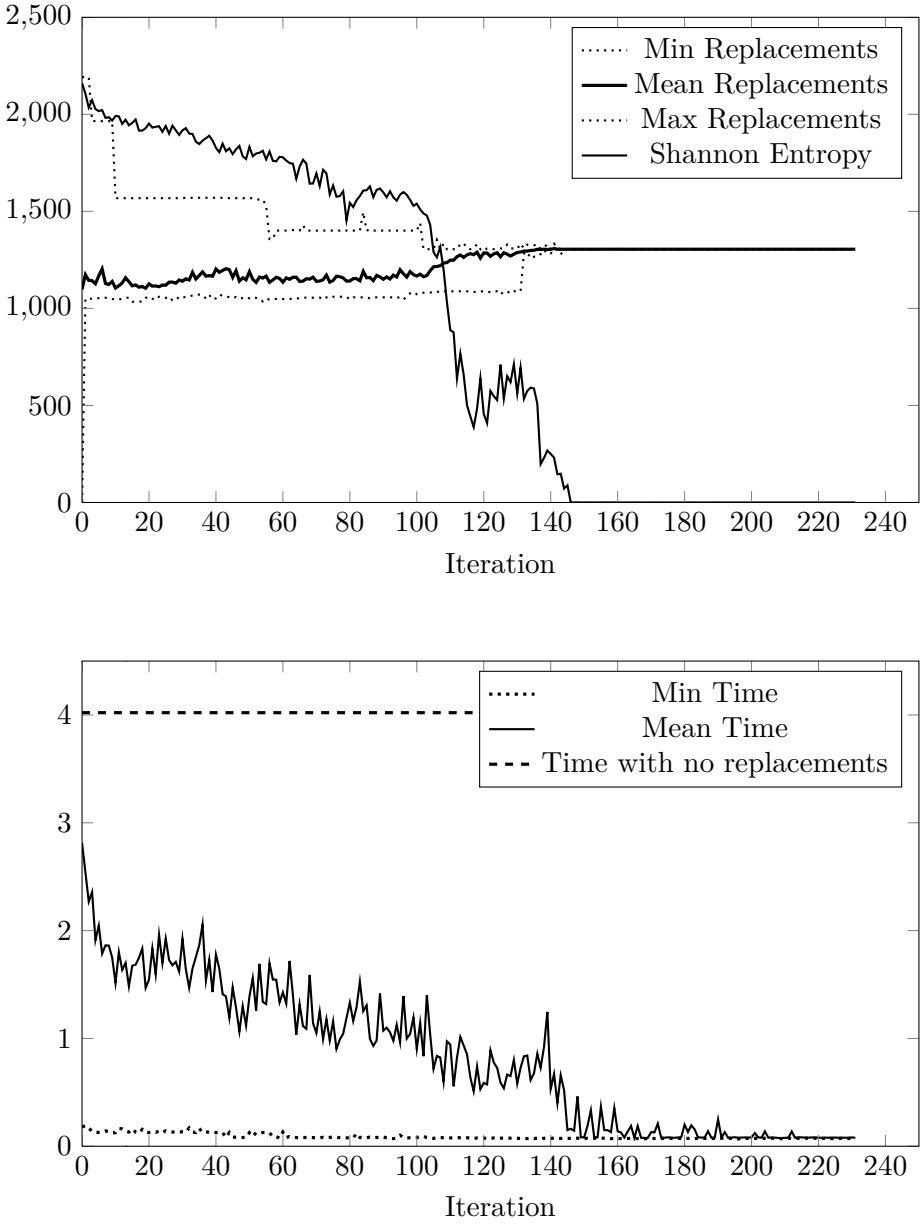


Figure 8: **square_1_false–unreach–call** (SAT): Replacements made (top), and time taken by MiniSat (seconds) (bottom) on each iteration.

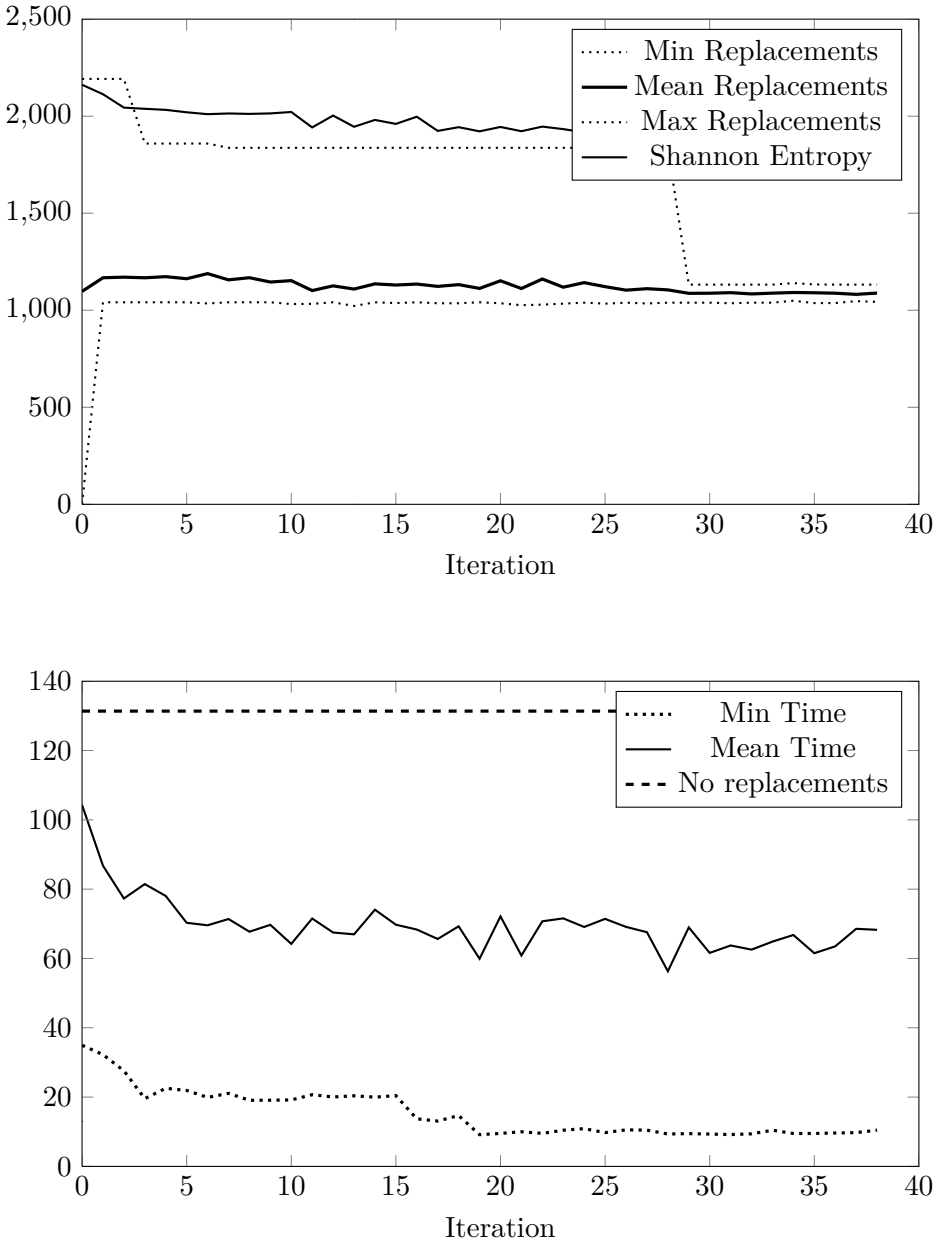


Figure 9: **square_7_true–unreach–call** (UNSAT): Replacements made (top), and time taken by MiniSat (seconds) (bottom) on each iteration.

and the entropy drops to almost zero. Between iterations 110 and 130, the entropy does increase a little, until the algorithm discards all but a very small range of replacements, and the entropy drops quickly as the algorithm finally converges. The average time taken by MiniSat also decreases steadily over the course of the run, before converging on a very strong improvement compared with the time without any preprocessing. Note that the best member of the initial population was already almost as good as the overall best. The performance in the best case is a dramatic improvement over the case with no replacements, but such a dramatic improvement on a SAT instance can, as mentioned above, be seen as the instance being tuned to the particular search heuristics employed by the SAT-solver, and does not necessarily represent a theoretically-interesting omptimisation.

Figure 9, on the other hand, shows a typical run of the genetic algorithm on an UNSAT benchmark. Initially, some improvement was made on the solutions found in the initial population, with the time for the overall best member being about half that of the best member in the initial population, but the Shannon entropy remains high throughout. The number of replacements being made does settle down to a narrow range, though, where about half of the possible replacements are made. It could be the case that, on these UNSAT instances, half of the replacements is the best that can be done, but it does not matter which half. On the other hand, a genetic algorithm may not be an appropriate way to find good sets of replacements on these instances, or the algorithm may just be configured incorrectly for these instances. However, despite not converging, the algorithm did find sets of replacements that lead to greatly improved MiniSat performance.

Overall, genetic algorithms worked well for finding good solutions in the SAT cases of these benchmarks, but less well in the UNSAT cases. However, simply taking a random sample of 50 sets of replacements in the first iteration produced some sets that outperformed the case where no replacements are made, even in UNSAT cases, and these were often improved by the genetic algorithm.

[This will tie into previous chapters:] We can therefore conclude that, yes, for these particular formulas, arising from software verification, there are generally sets of replacements that improve the performance of MiniSat. Additionally, random sampling of sets of replacements was sufficient to find improvements, and in SAT cases, a genetic algorithm was usually able to converge on good solutions.

7.2 Nature of solutions found

At the beginning of the genetic algorithm, we find the set of all those subclauses in the formula appearing in 30 or more clauses. Members of the population are then subsets of that set. The best performing subsets — across all of the instances — included about half of the subclauses, and good subsets were found very quickly.

First, it is important to note that, requiring membership of 30 or more clauses, these subclauses are already good candidates for replacement. Swapping out the subclause in at least 30 clauses and replacing it with a single variable — and then only adding a few clauses at the end — decreases the size of the formula overall. If the requirement for 30 or more clauses were not there, there would be many, many more possible replacements. In the worst case, a formula containing 2 clauses and n literals, with each clause containing all of the literals, would have n^2 possible replacements. Even simple formulas obtained from software verification problems can have millions of literals, so performing all — or even half — of the possible replacements would be infeasible.

The fact that these results show half of the subclases as the optimal number of replacements could be because of some particular feature of these formulas, which would suggest that further investigation of the encoding could be fruitful in finding an oracle for which replacements to make. On the other hand, it could be the case that half of the replacements simply represents the sweet spot between removing repeated subclauses and adding new clauses representing the extensions.

Very good sets of substitutions were usually found in the initial population of the genetic algorithm. That is, a random sample of 50 sets was sufficient to find a set that outperformed the empty set. Given that good sets are so easily found (at least on this type of formula), it seems likely that progress could be relatively easily made toward an oracle that facilitates obtaining such a subset before running the SAT-solver.

7.3 Toward an oracle

[Have results from trying half of subclauses on one formula]

[Somehow draw box and whisker diagram here of those results]

8 Conclusion and future work

References

[Agrawal et al., 1994] Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499.

[Audemard et al., 2010] Audemard, G., Katsirelos, G., and Simon, L. (2010). A restriction of extended resolution for clause learning SAT solvers. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 15–20.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., and Zhu, Y. (1999). Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM.

[Biere et al., 2014] Biere, A., Le Berre, D., Lonca, E., and Manthey, N. (2014). Detecting cardinality constraints in cnf. In *Theory and Applications of Satisfiability Testing–SAT 2014*, pages 285–301. Springer.

[Borgelt, 2003] Borgelt, C. (2003). Efficient implementations of apriori and eclat. In *FIMI'03: Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations*.

[Cook, 1976] Cook, S. A. (1976). A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397.

[D'silva et al., 2008] D'silva, V., Kroening, D., and Weissenbacher, G. (2008). A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178.

[Eén and Biere, 2005] Eén, N. and Biere, A. (2005). Effective preprocessing in sat through variable and clause elimination. In *Theory and Applications of Satisfiability Testing*, pages 61–75. Springer.

[Eén and Sörensson, 2004] Eén, N. and Sörensson, N. (2004). An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg.

[Esquivel et al., 1997] Esquivel, S., Leiva, A., and Gallard, R. (1997). Multiple crossover per couple in genetic algorithms. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 103–106. IEEE.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[Grefenstette, 1986] Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(1):122–128.

[Haken, 1985] Haken, A. (1985). The intractability of resolution. *Theoretical Computer Science*, 39:297–308.

[Kroening and Strichman, 2003] Kroening, D. and Strichman, O. (2003). Efficient computation of recurrence diameters. In *Verification, Model Checking, and Abstract Interpretation*, pages 298–309. Springer.

[Manthey, 2012] Manthey, N. (2012). Coprocessor 2.0–a flexible cnf simplifier. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 436–441. Springer.

[Marques-Silva and Sakallah, 1999] Marques-Silva, J. P. and Sakallah, K. A. (1999). Grasp: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521.

[Martins et al., 2011] Martins, R., Manquinho, V., and Lynce, I. (2011). Exploiting cardinality encodings in parallel maximum satisfiability. In *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, pages 313–320. IEEE.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM.

[Nicely, 2008] Nicely, T. (2008). Pentium FDIV flaw.

[Plaisted and Greenbaum, 1986] Plaisted, D. A. and Greenbaum, S. (1986). A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304.

[Rossi et al., 2006] Rossi, F., Van Beek, P., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423.

[Taherdangkoo et al., 2013] Taherdangkoo, M., Paziresh, M., Yazdi, M., and Bagheri, M. (2013). An efficient algorithm for function optimization: modified stem cells algorithm. *Open Engineering*, 3(1):36–50.

[Thierens and Goldberg, 1994] Thierens, D. and Goldberg, D. (1994). Convergence models of genetic algorithm selection schemes. In *Parallel problem solving from nature–PPSN III*, pages 119–129. Springer.

[Tseitin, 1983] Tseitin, G. S. (1983). On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg.

[Whitley et al., 1989] Whitley, L. D. et al. (1989). The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *ICGA*, volume 89, pages 116–123.

[Yan et al., 2015] Yan, Y., Gutierrez, C., Jn-Charles, J., Bao, F., and Zhang, Y. (2015). Accelerating SAT solving by common subclause elimination. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 4224–4225.

A Extended resolution proof of PHP for $n = 3$

The clauses representing the pigeonhole principle for $n = 3$ are:

$$\begin{array}{lll} \{P_{1,1}, P_{1,2}\}, & \{P_{2,1}, P_{2,2}\}, & \{P_{3,1}, P_{3,2}\}, \\ \{\neg P_{1,1}, \neg P_{2,1}\}, & \{\neg P_{1,1}, \neg P_{3,1}\}, & \{\neg P_{2,1}, \neg P_{3,1}\}, \\ \{\neg P_{1,2}, \neg P_{2,2}\}, & \{\neg P_{1,2}, \neg P_{3,2}\}, & \{\neg P_{2,2}, \neg P_{3,2}\}. \end{array}$$

We perform four extensions:

$$\begin{aligned} Q_{1,1} &\leftrightarrow P_{1,1} \vee P_{1,2}, \\ Q_{1,1} &\leftrightarrow P_{1,1} \vee P_{3,1}, \\ Q_{2,1} &\leftrightarrow P_{2,1} \vee P_{2,2}, \\ Q_{2,1} &\leftrightarrow P_{2,1} \vee P_{3,1}. \end{aligned}$$

This introduces the following additional clauses:

$$\begin{array}{lll} \{\neg Q_{1,1}, P_{1,1}, P_{1,2}\} & \{\neg P_{1,1}, Q_{1,1}\} & \{\neg P_{1,2}, Q_{1,1}\} \\ \{\neg Q_{1,1}, P_{1,1}, P_{3,1}\} & \{\neg P_{1,1}, Q_{1,1}\} & \{\neg P_{3,1}, Q_{1,1}\} \\ \{\neg Q_{2,1}, P_{2,1}, P_{2,2}\} & \{\neg P_{2,1}, Q_{2,1}\} & \{\neg P_{2,2}, Q_{2,1}\} \\ \{\neg Q_{2,1}, P_{2,1}, P_{3,1}\} & \{\neg P_{2,1}, Q_{2,1}\} & \{\neg P_{3,1}, Q_{2,1}\} \end{array}$$

Note that two of the clauses introduced in the extension step are actually duplicates.

The pigeonhole principle for $n = 2$ is represented by just three clauses:

$$\{Q_{1,1}\}, \{Q_{2,1}\}, \{\neg Q_{1,1}, \neg Q_{2,1}\}$$

The first two are easily derived with a handful of resolutions:

$$\dfrac{\dfrac{\{P_{1,1}, P_{1,2}\} \quad \{\neg P_{1,1}, Q_{1,1}\}}{\{P_{1,2}, Q_{1,1}\}} \qquad \{\neg P_{1,2}, Q_{1,1}\}}{\{Q_{1,1}\}}$$

$$\dfrac{\dfrac{\{P_{2,1}, P_{2,2}\} \quad \{\neg P_{2,1}, Q_{2,1}\}}{\{P_{2,2}, Q_{2,1}\}} \qquad \{\neg P_{2,2}, Q_{2,1}\}}{\{Q_{2,1}\}}$$

Then, making use of those clauses, we derive two lemmas. The first states, in terms of the mappings described in the section on extended resolution,

that if $\phi'(1) = 1$, then $\phi(3) = 1$, and the second states that if $\phi'(2) = 1$, then $\phi(2) = 2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{Q_{2,1}\}\quad\{\neg Q_{2,1}, P_{2,1}, P_{3,1}\}}{\{P_{2,1}, P_{3,1}\}}\quad\{\neg P_{1,1}, \neg P_{2,1}\}}{\{\neg P_{1,1}, P_{3,1}\}}\quad\{\neg P_{1,1}, \neg P_{3,1}\}}{\{\neg P_{1,1}\}}\quad\{\neg Q_{1,1}, P_{1,1}, P_{3,1}\}}{\{\neg Q_{1,1}, P_{3,1}\}^{(*)}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{Q_{1,1}\}\quad\{\neg Q_{1,1}, P_{1,1}, P_{3,1}\}}{\{P_{1,1}, P_{3,1}\}}\quad\{\neg P_{1,1}, \neg P_{2,1}\}}{\{P_{3,1}, \neg P_{2,1}\}}\quad\{\neg P_{2,1}, \neg P_{3,1}\}}{\{\neg P_{2,1}\}}\quad\{\neg Q_{2,1}, P_{2,1}, P_{2,2}\}}{\{\neg Q_{2,1}, P_{2,2}\}^{(**)}}$$

From those lemmas we finally derive the last of the clauses for $n = 2$.

$$\dfrac{\dfrac{\dfrac{\{\neg Q_{1,1}, P_{3,1}\}^{(*)}\quad\{\neg P_{1,1}, \neg P_{3,1}\}}{\{\neg Q_{1,1}, \neg P_{1,1}\}}\quad\{P_{1,1}, P_{1,2}\}}{\{\neg Q_{1,1}, P_{1,2}\}}\quad\dfrac{\{\neg Q_{2,1}, P_{2,2}\}^{(**)}\quad\{\neg P_{2,2}, \neg P_{1,2}\}}{\{\neg Q_{2,1}, \neg P_{1,2}\}}}{\{\neg Q_{1,1}, \neg Q_{2,1}\}}$$

The proof of the pigeonhole principle is completed from the clauses for $n = 2$ in a few steps, as shown in Section 3.

B Programs used to generate benchmarks with CBMC

Below I include three examples of the programs used to generate the benchmarks used in the genetic algorithm.

First, **sine__6__true–unreach–call**:

```
# 1 "sine.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 170 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "sine.c" 2
extern void __VERIFIER_error(void);
extern void __VERIFIER_assume(int);
# 27 "sine.c"
```

```
int main()
{
  float IN;
  __VERIFIER_assume(IN > -1.57079632679f && IN < 1.57079632679f);

  float x = IN;

  float result = x - (x*x*x)/6.0f + (x*x*x*x*x)/120.0f + (x*x*x*x*x*x*x)/5040.0f
      ;

  if(!(result <= 1.2f && result >= -1.2f))
    __VERIFIER_error();

  return 0;
}
```

Second, **square_3_false-unreach-call**

```
# 1 "square.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 170 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "square.c" 2
extern void __VERIFIER_error(void);
extern void __VERIFIER_assume(int);
# 27 "square.c"
int main()
{
  float IN;
  __VERIFIER_assume(IN >= 0.0f && IN < 1.0f);

  float x = IN;

  float result =
    1.0f + 0.5f*x - 0.125f*x*x + 0.0625f*x*x*x - 0.0390625f*x*x*x*x;

  if(!(result >= 0.0f && result < 1.39843f))
    __VERIFIER_error();

  return 0;
}
```

Third, **newton_2_8_false-unreach-call**

```
# 1 "newton.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
```

```
# 170 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "newton.c" 2
extern void __VERIFIER_error(void);
extern void __VERIFIER_assume(int);
# 33 "newton.c"
float f(float x)
{
  return x - (x*x*x)/6.0f + (x*x*x*x*x)/120.0f + (x*x*x*x*x*x*x)/5040.0f;
}

float fp(float x)
{
  return 1 - (x*x)/2.0f + (x*x*x*x)/24.0f + (x*x*x*x*x*x)/720.0f;
}

int main()
{
  float IN;
  __VERIFIER_assume(IN > -2.0f && IN < 2.0f);

  float x = IN - f(IN)/fp(IN);

  x = x - f(x)/fp(x);




  if(!(x < 0.1))
    __VERIFIER_error();

  return 0;
}
```