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

Eliot Ball

May 15, 2015

Summary

The summary.

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. 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 forsee some of the situations? Ideally, we would be able to prove beyond doubt that the system doesn't fail.

Bounded Model Checking is an approach to this 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. This formula is then checked to see whether it is satisfiable ("SAT") using a SAT-solver. If the formula is SAT, then there is a counterexample of the desired length, which may be obtained by inspecting the values of the variables in satisfying assignment. If the formula is not satisfiable (it is *unsatisfiable* or "UNSAT"), 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*. 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.

Modern SAT-solvers operate on a machine-readable representation of formulas in *conjunctive normal form* (CNF). A CNF formula is a conjunction of clauses, where each clause is a disjunction of literals:

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

Each literal is either a variable or its negation. Note that this formula just

requires one literal from each clause to be true.

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. Therefore SAT-solvers can effectively operate on all propositional formulas.

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 paths that would lead to failure for the same reason.

2 Encodings and pre-processing

For a given proposition about a system, there are many possible ways to encode that proposition in the CNF format accepted by the SAT-solver. For example, the ordering of the literals within each of the clauses, and the ordering of the clauses themselves may be changed. Because the SAT-solver has to make deterministic choices that are not prescribed in the DPLL algorithm, it may behave differently for different orderings. Therefore, even though $(l_0 \vee l_1) \wedge l_2 = (l_1 \vee l_0) \wedge l_2 = l_2 \wedge (l_0 \vee l_1)$, the SAT-solver may arrive at the solution to each of these formulas via a different sequence of steps.

Further, they may be multple ways to structure the representation of the

system as a propositional formula. Suppose we are writing a CNF formula that is the result of applying an associative binary relation R to three objects a, b and c. We could choose either (a R b) R c or a R (b R c). The SAT-solver may perform differently on each of these encodings. In real-world software verification, where we perform complex operations like multiplying and adding large binary numbers, there are many, many possible encodings, some of which perform drastically better than others. For example, a binary multiplier encoded as a table recording the result for every possible pair of operands performs much worse than one which borrows ideas from optimised hardware multipliers. [Is this true?]

The problem of picking an encoding for a proposition 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. If the time saved by the SAT-solver is greater than the time taken to detect and apply the optimisations, then a net gain is made.

3 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 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. $a \vee \neg b = \{a, \neg b\}$), the resolution rule works as follows.

1	$C_1 \cup \{L\}$	premise
2	$C_2 \cup \{\neg L\}$	premise
3	$C_1 \cup C_2$	res **1**, **2**

For example, we can use resolution to prove the pigeonhole principle for the easy $n = 2$ case:

1	$\{P_{1,1}\}$	premise
2	$\{P_{2,1}\}$	premise
3	$\{\neg P_{1,1}, \neg P_{2,1}\}$	premise
4	$\{\neg P_{2,1}\}$	res **1**, **3**
5	$\bot$	res **2**, **4**

This is, in fact, a *tree resolution* proof, because no clause is antecedent to 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. 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. 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. 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 suggests itself as a good candidate for a more powerful system because it permits a polynomial-length proof of the pigeonhole principle [Cook, 1976].

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 phrase 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

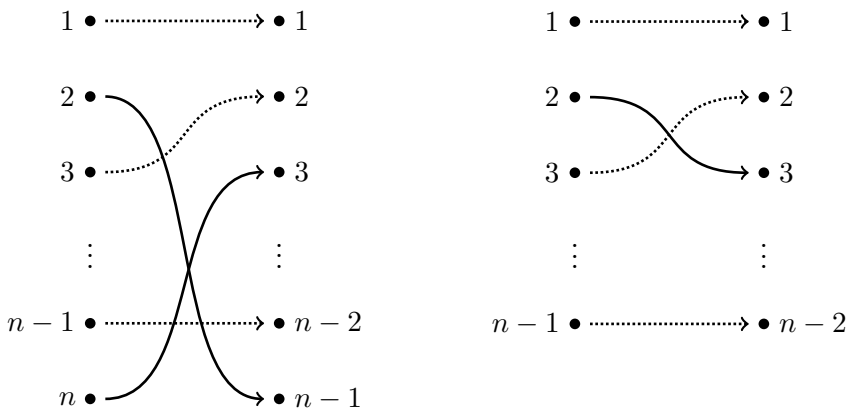
$$\phi'(i) = \begin{cases} \phi(n) & \text{if } \phi(i) = n-1, \\ \phi(i) & \text{otherwise} \end{cases}$$

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 1). 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.

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 1: 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:

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

These clauses describe the pigeonhole principle for $n-1$ items. Repeating this process of extension and resolution $O(n)$ times 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 which, 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 some cases.

For this project, I hypothesise that, for some propositional formulas arising in real-world model checking, there are sets of substitutions of repeated subclauses with size 2 or 3 (that is, there are pairs or triplets of literals which appear in multiple clauses) which can be replaced by extension with a single variable, leading to an improvement in the performance of the MiniSat solver on those instances. I investigate this by attempting to use a genetic algorithm to find such sets of substitutions.

5 Genetic algorithms

A *genetic algorithm* is a search algorithm that mimics natural selection. In the most basic form of the algorithm, sufficiently long binary strings are mapped onto 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 nubmer 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 (See Figure 2):

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 $m = a[0 \ldots k) \parallel b[k \ldots n)$.
3. **Mutation:** Randomly flip some of the bits in m and then add it to the new population.

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

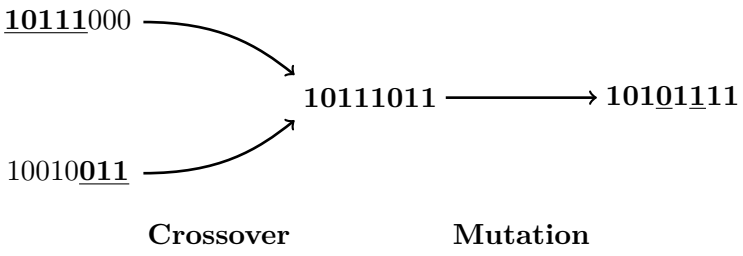


Figure 2: An illustration of the generation of a new member from a pair of parents in a genetic algorithm

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 k members, for some k. 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 k times, where k 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.

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. 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 another half 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 3). The constructor takes as an argument a CNF formula in the standard DIMACS format (See Figure **??**), and stores it internally as a list of lists of integers. Each inner list represents a clause, and each integer represents a literal. The 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 4). 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 in 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).

```
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 3: 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.)

```
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 4: 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 5: The functions used to run the genetic algorithm on the subclause sets. (Code that performs logging has been omitted.)

The run is set up by three functions that make use of the `Formula` and `Population` classes (See Figure 5). 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 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 hasnt' changed more by more than 0.5% between each of three consecutive iterations, and then returns the fittest member of the final population.

7 Analysis of results

- Analysis

8 Conclusion

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.

[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.

[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.

[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.