# Solving Satisfiability with Ant Colony Optimization and Genetic Algorithms

Dominik Moritz and Matthias Springer
{firstname.lastname}@student.hpi.uni-potsdam.de

University of Potsdam, Institute for Computer Science

**Abstract.** In this paper, we present our implementation of a SAT solver that uses a genetic algorithm and an ant colony algorithm. We show how we significantly increased the performance of these algorithms using preprocessing steps like factorization and using optimizations for escaping from local optima. The genetic algorithm achieves this by forcing variables and triggering catastrophes. The ant colony algorithm uses a most constrained variable heuristic and blurs pheromones values. Finally, we evaluate the overall performance of our implementation.

## 1 SATISFIABILTY solvers

The *boolean satisfiability problem* or SAT, is the problem of finding an assignment for variables that satisfy a boolean formula. In other words, with the assignment the formula evaluates to *true*.

The SAT problem was the first known example of an NP-complete problem [2]. It is widely believed (but not yet proven) that no algorithm exists that solves all instances of SAT deterministically in polynomial time. There exists a large number of practical problems that are NP-complete such as planning, hardware verification or theorem proving. All problems in NP can be reduced to SAT or to any other NP-complete problem in polynomial time. Solving just one NP-complete problem efficiently solves all problems in NP efficiently. Therefore, NP-complete problems and SAT in particular are among the most relevant problems in computer science.

Even though there currently exists no SAT solver that can solve all instances efficiently, a variety of algorithms have been developed. The first class of algorithms are *conflict-driven clause learning* (CDCL) algorithms, which extend the idea of the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [5]. These algorithms are complete algorithms, which means that they, given enough time, are guaranteed to find a solution. The second class of algorithms are *stochastic optimization algorithms*. These incomplete algorithms cannot guarantee to find a solution but may find a solution faster than a complete algorithm. In this paper, we will show how genetic algorithms and ant colony algorithms can be used to solve SAT instances. What all algorithms have in common is that they explore the search space of possible assignments, gradually improving the solution and eventually hitting an assignment that makes the problem evaluate to true.

However, due to the nature of these algorithms, there is no guarantee that the algorithms will find a solution even though it may exist.

## 2 Previous work

Dorigo proposed ant colony optimization as a metaheuristic for solving a wide set problems [6]. A metaheuristic is an abstract algorithm that describes the steps that need to be carried out for all problems, such as deciding whether to accept a solution candidate. MAX-MIN Ant System [13] and Ant Colony System [7] are among the first implementations for solving the Traveling Salesman Problem. Sethuram and Parashar described how to solve SAT for the design of VLSI circuits using ant colony optimization [12]. Villagra and Baran described optimizations like stepwise adaptation of weights for clauses and refining functions [14].

There is a high number of publications about the application of evolutionary algorithms to the SAT problem such as [10][9]. O. P. Cruz and A. Cruz summarized different selection and mutation schemes for genetic algorithms [3]. H. Ellerweg has published a study of different selection and mutation strategies [1].

## 3 Formats and data structures

In this section, we describe the data structures that are used in both the genetic algorithm and the ant colony optimization algorithm.

### 3.1 Boolean formulas in conjunctive normal form

A formula in conjunctive normal form (CNF) consists of a set of clauses. To satisfy the formula, every clause must be satisfied, i.e. every clause must have at least one literal that is satisfied.

In the DIMACS format, every clause is represented as a set of signed integers. A negative value represents a negated variable. For instance, `1 -5 4` stands for the clause $x_1 \vee \neg x_5 \vee x_4$.

We transform clauses into *full clauses* in order to simplify satisfiability tests. A full clause for a formula with $n$ variables is a matrix $c \in \mathbb{B}^{n \times 2}$. The $i^{\text{th}}$ column contains truth values for $x_i$ in $c_{i,1}$ and $\neg x_i$ in $c_{i,2}$. Consequently, $c_{i,1}$ and $c_{i,2}$ can never be true ($\top$) at the same time, but they are both false ($\bot$) if the variable is not part of the formula. Figure 1 shows how the DIMACS clause `1 -5 4` is written as a full clause.

$$\begin{bmatrix} \top & \bot & \bot & \top & \bot \\ \bot & \bot & \bot & \bot & \top \end{bmatrix}$$

**Fig. 1.** Full clause representation for $x_1 \vee \neg x_5 \vee x_4$ and 5 variables.

### 3.2 Solution candidates

We call a binding for every variable a solution candidate. Only if we know that the binding satisfies the boolean formula we call it a solution or model.

Solution candidates are typically represented as an enumeration of truth values. For instance, $[\top, \bot, \bot, \top, \top]$ denotes the solution candidate $\{x_1, \neg x_2, \neg x_3, x_4, x_5\}$.

Before we test if a solution candidate satisfies a formula, we transform it into a *full candidate*, similar to the full clauses described in Section 3.1. A full candidate is a matrix $s \in \mathbb{B}^{n \times 2}$. The $i^{\text{th}}$ column contains truth values for $x_i$ in $c_{i,1}$ and $\neg x_i$ in $c_{i,2}$. Consequently, either $s_{i,1}$ xor $s_{i,2}$ must be true, since the variable must be either satisfied or unsatisfied. Figure 2 shows an example for a full solution candidate.

$$\begin{bmatrix} \top & \bot & \bot & \top & \top \\ \bot & \top & \top & \bot & \bot \end{bmatrix}$$

**Fig. 2.** Full solution candidate representation for $\{x_1, \neg x_2, \neg x_3, x_4, x_5\}$.

### 3.3 Evaluating a solution candidate

Testing if a solution candidate satisfies a formula is done very often in all our algorithms. Therefore, this step must be performed quickly.

Formulas are represented as a list of full clauses. Effectively, a formula is a matrix $f \in \mathbb{B}^{m \times n \times 2}$, where $m$ is the number of clauses and $n$ is the number of variables. To test a solution candidate $c$, we calculate $f_i \wedge c$ bitwise for every $1 \leq i \leq m$ (for every clause). $c$ satisfies $f$ iff every resulting matrix contains at least one $\top$ value. Figure 3 shows how a full candidate is tested efficiently with NumPy.

```
1 def evaluate_full_candidate(f, c):
2     return np.any(f & c, axis=(2, 1))
```

**Fig. 3.** Evaluation of full candidate with NumPy

## 4 Preprocessing and Optimizations

In this chapter, we discuss preprocessing and optimization techniques that can speed up the algorithms presented in the next chapters. We will take take a look at how to support multiprocessor systems.

### 4.1 Removing clauses and unconstrained variables

Before an instance is passed to an algorithm, unconstrained variables, i.e. variables that are not used in any clause, are removed. Furthermore, clauses with only one literal are removed because it is obvious how to assign the variable. All other clauses that contain this variable can be simplified, too. If the variable in the clause has the same sign (positive or negated) we remove the whole clause because it is now satisfied. Otherwise, we simply remove the variable from the clause. An instance is unsatisfiable if we end up with an empty clause. An instance is satisfiable if we end up with no clauses.

### 4.2 Configuration Profiles

The performance of the algorithms discussed in the next chapter heavily relies on the choice of certain parameters such as the seed value for random numbers. We found out that there are some *good* parameter combinations that perform well on certain instances. However, the same parameters combinations may perform badly with different instances. We call a set of parameters a profile.

To gain advantage of multiprocessor systems and to be able to solve different types of instances fast, the same instance can be processed with different profiles simultaneously. Therefore, the program spawns a user-defined number of worker processes. All processes are terminated as soon as the first process returns with a solution.

### 4.3 Factoring variables

Generating new instances with a set of variable assignment assumptions is an alternative to spawning new processes based on configuration profiles. Factoring an instance by the variable $x_i$ generates two *factored instances* with the additional clauses $x_i$ and $\neg x_i$ respectively[1]. Each factored instance is preprocessed and then run in a separate worker process. The original solution is unsatisfiable if both factored instances are found unsatisfiable during preprocessing.

Factored instances can again be factored by another variable. The total number of factored instances grows exponentially with the number of factorings. Therefore, if $c$ is the number of processing units, we should not factor by more than $\log_2 c$ variables.

We always factor by the most constrained variable, i.e. the variable that appears in the highest number of clauses. Our implementation combines configuration profiles and factoring variables. All factored instances are run with the default profile. If the user has chosen to use more processes than factored instances were generated, random factored instances are run with random profiles. If a process stopped because the maximum iterations limit was reached, a

[1] Factoring a variable is equivalent to the splitting step of the Davis-Putnam algorithm.

factored instance is chosen round robin and run with a random profile and more iterations[2].

# 5 Genetic Algorithm

Darwin's evolutionary theory of natural selection proposed in *The Origin of Species*[4] states that variability in a natural population as a result of mutation and new, sexually produced gene combinations help a population evolve and assure its survival. Genetic algorithms [8] or *GAs* mimic evolution and due to their robustness they are used in a wide variety of applications.

When implementing a genetic algorithm, we should try to build an algorithm that converges quickly while preserving an acceptable genetic diversity. The second property is very important for the SAT problem because a solution that satisfies all clauses but one is not necessarily similar to a solution that satisfies all clauses.

Whether a solution is found or not largely depends on the appropriate settings especially of the population size, selection and mutation rate. Consequently, using profiles where the combination of parameters can be coordinated as discussed in 4.3 significantly improves the performance of a GA.

## 5.1 Biological background and terms

An *individual* consists of cells that have the same set of *chromosomes*. A chromosome is an organized structure of DNA which encodes a model of the whole organism and is composed of *genes*, each of which encodes a specific *trait*. All chromosomes together form the *genome*. Individuals together form a *population* which *reproduces*. During this process, genomes of several individuals are recombined. This process is called *crossover*. The new individual's genome, or offspring, is not the exact recombination of the parent's genomes but slightly different. The process that changes the genome is called *mutation* and is caused by errors during the crossover. Some individuals die, some survive and perform poorly, and others perform better and have a smaller likelihood of dying. This performance is called *fitness*.

## 5.2 Basic algorithm and genetic operators

Figure 4 shows the basic algorithm that we use in our implementation. However, keep in mind that our implementation is modified in a way that allows customization (e.g. profiles) and has code optimizations.

The fitnesses of the newly created offspring is recalculated in each iteration. The fitness function calculates the number of satisfied clauses, as described in Section 3.3. In the selection step, the algorithm can either choose from all

[2] The program does not terminate if the instance is unsatisfiable, unless detected during preprocessing.

```
def run():
    population = generate_polpulation()
    while True:
        selection = get_selection(population)
        offspring = create_offspring(selection)
        mutate_offspring(offspring)
        # overwrite chromosomes in population but
        # do not overwrite elites
        replace_no_elites(population, offspring)
        fitnesses = calculate_fitnesses(population)
        if num_clauses in self.fitnesses:
            return best(population, fitnesses)

def create_offspring(selection):
    offspring = []
    for parents in selection:
        offspring.append(crossover(parents))
    for _ in range(parameters['Random offspring']):
        offspring.append(random_chromosome())
    return offspring
```

**Fig. 4.** Basic genetic algorithm

non-elites randomly or with a probability that correlates to the fitness of the chromosomes. The second way of selecting chromosomes for forming offspring leads to faster convergence but may reduce diversity. We use one point crossover to breed offspring from the selection. Elites are a fixed number of chromosomes with the best performance.

### 5.3 Parameters

Adjusting the parameters for the different algorithms can dramatically change the performance and behavior, as we will see in Section 7.

- *Population size:* The number of individuals in a population at any given time. The higher the number, the more variability is possible. However, a larger population increases the computation time of every iteration.
- *Selection rate:* Share of the population that is selected for breeding offspring. We use *roulette-wheel selection* or random selection.
- *Mutation Rate:* The probability under which mutation is applied to the genotype of an offspring individual.
- *Forced chromosomes:* Number of chromosomes where missing clauses are forced to become true. See Section 5.6.
- *Seeded chromosomes:* Number of chromosomes that are created with a simulated annealing algorithm [11] with a better performance than the average chromosome.

- *Random offspring:* Number of offspring chromosomes that are created at random instead of by crossover.
- *Select with probability:* Selection can be done randomly or with a probability that corresponds to the individual's fitness. Selecting randomly is much faster but the algorithm converges slower.

### 5.4 The fitness dilemma

The fitness of a chromosome guides the search in the genetic algorithms. It decides which chromosomes survive or which are selected for crossover. However, especially for the SAT problem it cannot be guaranteed that an increase of fitness leads to a, or sometimes the only, solution. We will discuss the effects of this problem in Section 5.5 and in Section 7.4.

### 5.5 Genetic algorithm optimizations

In this section, we cover three possible optimizations that either improve the convergence-diversity ratio or increase the algorithm's performance by adding knowledge about the problem to algorithm.

*Increasing diversity by adding new offspring randomly.* This technique adds random chromosomes to the population in every single iteration. Even though this may sound like a good idea, it does actually decrease the performance in the case of SAT because these new chromosomes are generally too bad compared to the average chromosome and are replaced too quickly.

*Avoiding premature convergence with catastrophes.* A way to avoid premature convergence and reduced diversity are catastrophes that replace large parts of the population with random chromosomes. A catastrophe either happens after a certain number of iterations or when the diversity becomes too low. We define the diversity of a population as the standard deviation. This optimization yields very good results for some of the SAT problems. Figure 5 shows how catastrophes affect the distribution of chromosome performance. In a catastrophe (at about 140, 240 and 340 iterations), the non-elite part of the population is replaced by random chromosomes which increases the diversity but does not affect the performance of the best chromosome.

### 5.6 Future work

Possible optimizations that we considered but have not yet implemented include selection of mutated genes based on previous mutations of the same chromosome or based on the effect it would have on the individual's performance. Also, other crossover strategies such as *uniform crossover* could be tested.

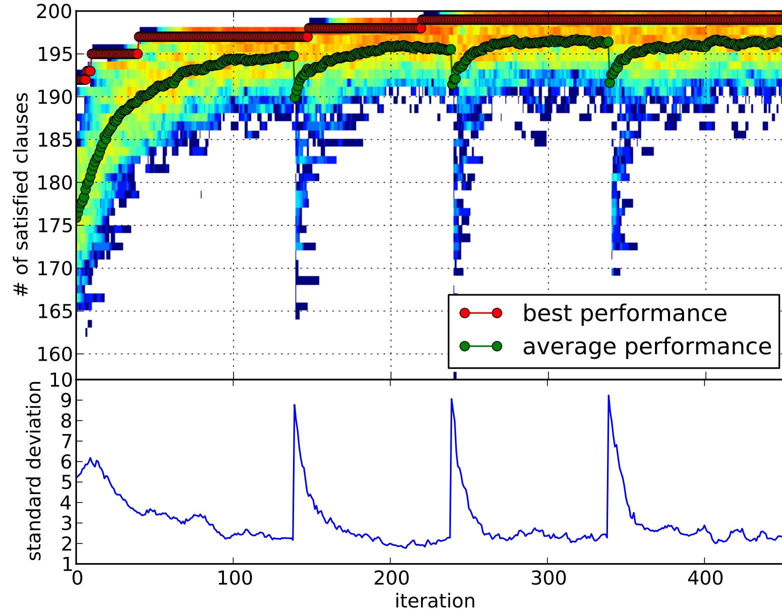



**Fig. 5.** Chromosome performance, affected by catastrophes. The color map shows how many chromosomes with a certain performance exist.

*Forcing clauses to become satisfiable.* The algorithm can be configured to make a number of unsatisfied clauses true by inverting one literal from the clause. This optimization was introduced when we noticed that the GA often fails to satisfy the last clause because it has no knowledge about the problem. This optimization increases the performance of the GA for some problems significantly.

## 6 Ant Colony Optimization

In this section, we present our implementation of the ant colony optimization algorithm. Villagra and Baran suggested to use adaptive fitness functions in order to evade local optima [14]. We adopted their idea of clause weight adaption and added the concept of blurring pheromones, that we present in chapter 6.5.

### 6.1 Finding shortest paths with pheromone trails

Ant colony optimization simulates the natural behavior of multiple ants that try to find a short path from the anthill to a place where food was found. On their way back to the anthill, ants produce pheromones that evaporate after some time. On their way to the food and back to the anthill, ants take paths that

contain a high quantity of pheromones with a high probability. However, they still take paths with fewer or no pheromones with a low probability, in order to find a shorter path.

An ant can take a shorter path in a shorter period of time. Therefore, when an ant heads for the food a second time, the quantity of pheromones on this path is still higher than the quantity of pheromones on a longer path, because a higher quantity of pheromones already evaporated on the longer path. In the end, short trails tend to have a greater quantity of pheromones present.

The quantity of pheromones increases more the more ants take that path. However, when a lot of ants initially take a longer path, the colony might be trapped and unable to find a shorter path, because the long path already has a high quantity of pheromones present.

### 6.2 Finding proper variable assignments with pheromone values

We now adapt the concept of pheromone trails for finding shortest paths to finding variable assignments that maximize the number of satisfied clauses. For each variable $x_i$, $1 \leq i \leq n$ we maintain two pheromone values $ph_{x_i}$ and $ph_{\neg x_i}$.

A run of the ant colony optimization algorithm consists of several iterations. In each iteration, a number of ants is simulated subsequently. Every ant generates a solution candidate by choosing $n$ literals. Let $\mathbb{L}$ be the set of all literals with $|\mathbb{L}| = 2n$. A literal $l \in \mathbb{L}$ is chosen with a probability of $\text{prob}(l)$.

$$\text{prob}(l) = \frac{ph_l}{\sum_{m \in \mathbb{L}} ph_m}$$

**Fig. 6.** Probability for choosing a literal $l$.

To simulate the evaporation of pheromones, we reduce all pheromones by a constant percentage. Afterwards, we evaluate the quality of every solution generated in the current iteration, i.e. the number of satisfied clauses per solution candidate. Then we update the pheromone values based on the best solution candidate by adding the number of solved clauses to the pheromone value for every literal that was chosen. For instance, if the best solution $\{x_1, \neg x_2, \neg x_3\}$ satisfied five clauses we add 5 to $ph_{x_1}$, $ph_{\neg x_2}$ and $ph_{\neg x_3}$.

### 6.3 Most constrained variables heuristic

The idea of the MCV (most constrained variable) heuristic is that more constrained variables are more important than less constrained variables because a highly constrained variable can satisfy a lot of clauses all at once. Therefore, we change the probability function for choosing literals as follows.

Figure 8 shows the modified probability function. $cstr_l$ is the constrainedness of the variable of literal $l$, i.e. the number of clauses that contain the variable of

```
1  pheromones = [PH_MAX] * l

2  def run():
3      while True:
4          best_literals, best_evaluation = None

5          for a in range(NUM_ANT):
6              literals = choose_literals()
7              evaluation = evaluate_solution(literals)

8              if evaluation == NUM_CLAUSES:
9                  # found solution
10                 return literals
11             elif evaluation > best_evaluation:
12                 best_literals, best_evaluation = literals, evaluation

13         update_pheromones(best_literals, best_evaluation)

14 def choose_literals():
15     # choose NUM_VAR from all literals,
16     # each with probability pheromones[literal]

17 def evaluate_solution(literals):
18     return np.sum(evaluate_candidate(chosen))

19 def update_pheromones(best_literals, best_evaluation):
20     pheromones[best_literals] += best_evaluation
```

**Fig. 7.** Basic ant colony optimization algorithm

$l$. $\alpha$ and $\beta$ are exponential factors that control the influence of the pheromones and the MCV heuristic[3].

### 6.4 Weight adaptation heuristic

The idea of the weight adaption heuristic is to make clauses, that turned out to be hard to solve, more important during evaluation. From time to time[4],

[3] These factors are called `EXP_PH` and `EXP_MCV` in the Python code.

[4] In the Python code, `WEIGHT_ADAPTION_DURATION` is the number of evaluations until clause weights are changed.

$$\text{prob}(l) = \frac{ph_l^{\alpha} \cdot cstr_l^{\beta}}{\sum_{m \in \mathbb{L}} ph_m^{\alpha} \cdot cstr_m^{\beta}}$$

**Fig. 8.** Probability for choosing a literal $l$ with MCV heuristic.

we examine all unsatisfied clauses during evaluation and increase their *clause weight*. Initially, all clauses have the same clause weight.

```
def evaluate_solution(literals):
    candidate_counter += 1

    solved_clauses = evaluate_candidate(chosen)
    num_solved_clauses = np.sum(solved_clauses)
    evaluation = np.sum(solved_clauses * clause_weights)

    if candidate_counter == WEIGHT_ADAPTION_DURATION:
        clause_weights += ~solved_clauses
        candidate_counter = 0

    return evaluation, num_solved_clauses
```

**Fig. 9.** Evaluation of solution candidates with weight adaption heuristic.

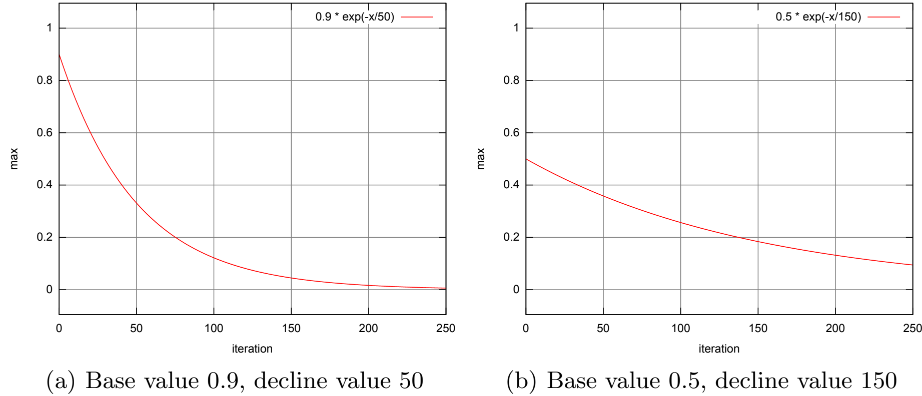
Figure 6.4 shows how we changed the evaluation function. We now return two values, the quality of the solution candidate (`evaluation`) and the number of solved clauses (to check if we found a solution). The quality of the solution candidate is sum of all weights of solved clauses. For instance, if a clause has a weight value of 5 and it was solved then this clause contributes 5 to the quality value. In line 7, we increase the weight of all unsatisfied clauses by 1.

### 6.5 Blurring pheromones

The idea behind simulated annealing [11] is to accept worse solution candidates with a probability that decreases with the number of iterations, in order to escape from local optima. For the same reason, our implementation of the ant colony optimization algorithms blurs pheromone values. To each pheromone value $ph_l$ the value $r \cdot ph_l$ is added, where $r$ is a random number with $-max < r < max$. In contrast to simulated annealing, we do not accept or reject certain solutions but add random changes to solution candidates. As Figure 10 shows, *max* decreases with each iteration. This is similar to the cooling process of simulated annealing that lowers the temperature slowly.

$$max(i) = b \cdot e^{\frac{-i}{d}}$$

**Fig. 10.** Maximum blurring during iteration $i$ with base value $b$ and decline value $d$.




(a) Base value 0.9, decline value 50

(b) Base value 0.5, decline value 150

**Fig. 11.** *max* with two different configurations. In the first iteration, *max* is the base value. Then *max* decreases. Lower decline values cause *max* to decrease faster.

In our test cases, blurring was also effective when it was not used after every iteration but only from time to time[5].

# 7 Benchmarks

We ran all benchmarks on a desktop computer with an Intel Core i5 750 processor[6] and 8GB main memory. We used Ubuntu 12.04[7] and Python 2.7.3 with NumPy 1.7.0 and Bottleneck 0.6.0.

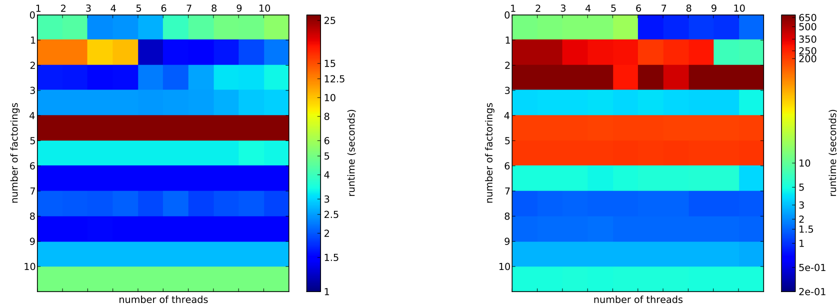
We used a seed value for the random number generator to ensure that the results are reproducible. Some of the benchmarks use more than one process. Each process has its own random number generator. Therefore, the generation of random number does not depend on process scheduling and is reproducible.

## 7.1 Overall performance

Figure 12 shows the overall performance of the ant colony optimization algorithm and the genetic algorithm. We ran both algorithms with different number of factorings and numbers of processes combinations. Algorithm-specific parameters like *number of ants* were chosen automatically by the algorithm.

*Overall performance.* For most parameter combinations, the ant colony optimization algorithm is faster than the genetic algorithm.

[5] In the Python code, `BLUR_ITERATIONS` is the number of iterations until the pheromones are blurred.
[6] 4 cores, 2.67GHz each, no hyperthreading
[7] Kernel version 3.2.0-31.




(a) Ant colony optimization algorithm.

(b) Genetic algorithm.

**Fig. 12.** Runtime for `random_ksat11.dimacs` with different number of factorings and number of processes combinations. All runtime values for the genetic algorithm are limited 700 seconds, i.e. some parameter combinations took more time but are shown as 700 seconds. Runtime values are the arithmetic average of 3 runs.

*More factored instances than CPU cores.* The number of factored instances grows exponentially with the number of factorings. Nevertheless, the overall performance can sometimes be increased with a high number of factorings, even if more factored instances are generated than processing units are available. For instance, the ant colony optimization with 8 factorings generates 256 factored instances of which 242 instances are found unsatisfiable during preprocessing. The remaining 14 instances were run simultaneously on a computer with four CPU cores. Due to the preprocessing, these 14 instances could be solved faster than 4 unfactored instances that were run simultaneously with different profiles (4 processes, no factorings).

*Factoring anomalies.* Both algorithms show runtime anomalies at certain factoring numbers. The ant colony optimization algorithm performs badly with 4 factored instances, while performing much better with fewer or more factored instances. Similarly, the genetic algorithm performs badly with 1, 2, 4 or 5 factored instances. In general, we expect the runtime to double with every additional factoring if no factored instances are found unsatisfiable and more factored instances were generated than processing units are available. In our test environment, this is the case when changing from 2 factorings to 3 factorings (ant algorithm). However, the runtime for the ant colony optimization algorithm increases by a factor of 8. It seems that the generated factored instances showing the anomaly are harder to solve than the original instance. Therefore, we have to keep in mind that additional factorings can also worsen the performance significantly.

*Time spent on preprocessing.* We expect that the time spent on preprocessing increases exponentially with the number of factorings because the number of factored instances increases exponentially with the number of factorings. In

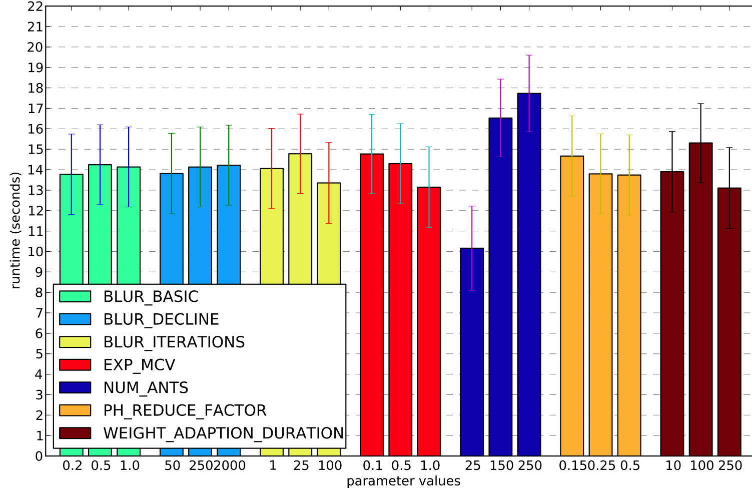



**Fig. 13.** Performance of the ant colony optimization algorithm with different parameter combinations for `random_ksat11.dimacs`. Each bar shows the geometric average runtime of $3^6 = 729$ runs with a single fixed parameter and all possible combinations of the remaining shown parameter values (3 per parameter). The small line on top of each bar shows the standard deviation, divided by 10. Runs were aborted after 45 seconds (accounted for 45 seconds). The parameter values are mostly based on findings in other papers, particularly [14].

contrast, the time spent on running the actual algorithm can decrease if the factored instances are easier to solve. Beginning with 6 factorings, the time spent on preprocessing is equal to or greater than the time spent on running the ant colony optimization algorithm on all factored instances. Therefore, increasing the number of factorings further cannot speed up the algorithm by more than a factor of 2 (if the whole runtime is spent on preprocessing). Actually, the overall performance worsens when increasing the number of factorings further.

### 7.2 Ant colony optimization parameters

Figure 13 shows the influence of parameters on the ant colony optimization algorithm's performance. We did not use preprocessing or factoring for these benchmarks. For every parameter, we selected 3 different promising values. We ran the algorithm with every possible parameter combination, resulting in 2187 runs. Every bar shows the geometric average runtime of all runs with the corresponding parameter value.

*High standard deviation.* All parameter values show a high standard deviation. We had very slow and very fast runs for all parameter values. For instance,

some runs finished after less than 0.5 seconds whereas some runs timed out with almost the same parameter configuration. Nevertheless, it is obvious that some parameter configurations perform significantly better on average.

In Section 7.4, we analyze the choice of the seed parameter on the algorithms' performance. The same results apply to other parameters, too, because every parameter change triggers a different random number sequence.

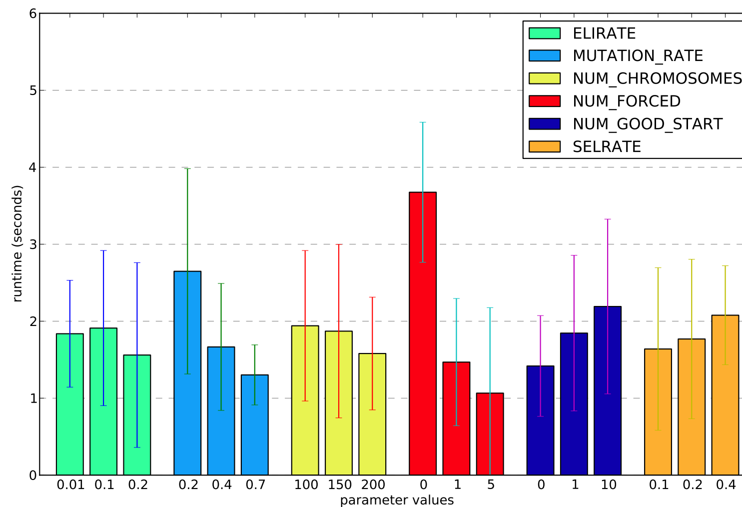



**Fig. 14.** Performance of the genetic algorithm with different parameter combinations for `random_ksat3.dimacs`. Each bar shows the geometric average runtime of $3^5 = 243$ runs with a single fixed parameter and all possible combinations of the remaining shown parameter values (3 per parameter). The small line on top of each bar shows the standard deviation, divided by 10. Runs were aborted after 45 seconds (accounted for 45 seconds).

*Influence of parameters.* The number of ants per iteration has the greatest influence on the runtime, because by simulating more ants more solutions are generated and evaluated. For every iteration, the solution candidate of the best ant is used for updating pheromone values and probabilities. The other solution candidates are rejected. Therefore, we should keep the number of ants small to guarantee a fast progress of the algorithm.

It is interesting that the base value and the decline value for blurring pheromones have little effect on the performance of the algorithm. However, the choice of the blurring frequency (number of iterations between blurrings) is important.

We are surprised that the pheromone reduce factor has such little influence on the algorithm's performance. We expected it to have much more influence

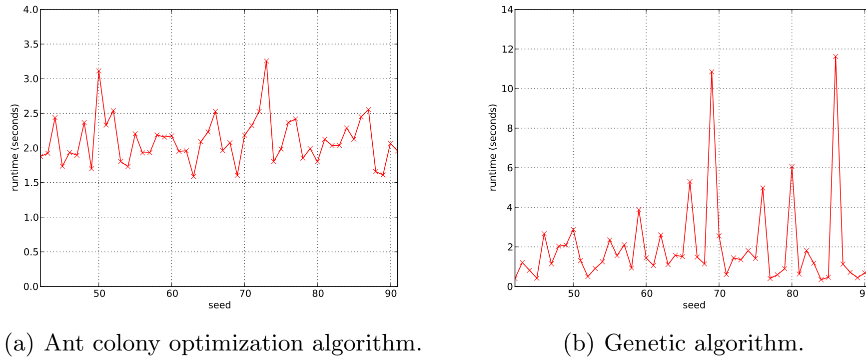
on the algorithm's performance than the parameters for blurring pheromones because it controls the evaporation of pheromones in *every* iteration. Blurring only takes place during some iterations and in most test runs – when the basic value was 0.2 or 0.5 – the average blurring factor is smaller than a pheromone reduce factor of 0.25 or 0.5.

The exponential factor for the most constrained variable heuristic and the weight adaption parameter are also crucial.

### 7.3 Genetic algorithm parameters

Figure 14 shows the influence of parameters on the genetic algorithm's performance. The way the figures have been calculated is the same process used in Section 7.2.

We can see that that the number of forced clauses and the mutation rate are the most important parameters for the benchmarked instance.




(a) Ant colony optimization algorithm. (b) Genetic algorithm.

**Fig. 15.** Runtime for `random_ksat3.dimacs` with 50 different seed values.

### 7.4 Seed sensitivity

We examined how much the performance of an algorithm changes depending on the value of the random seed. As can be seen in Figure 15, the ant colony algorithms shows little reaction to varying seeds but the GA's performance depends heavily on the choice of the seed.

A high difference in the runtimes depending on the seed should generally be avoided since it is an indication of premature convergence. Because of that, as discussed in Section 5.5, we applied measures to avoid this behavior. As can be seen in Figure 16, changing the mutation rate does not significantly affect how much the performance depends on the seed. A relatively low selection rate however, has a positive effect on the robustness of the algorithm. Another

measure that we used to increase the robustness of the genetic algorithm are catastrophes. This optimization even benefits from a high convergence.

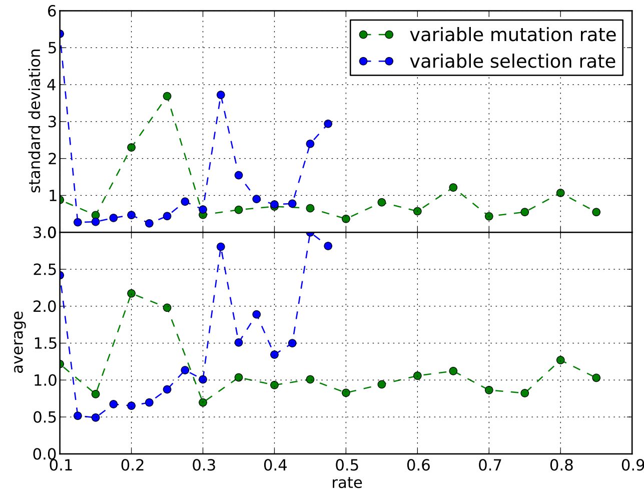



**Fig. 16.** Effect of changing mutation rate or selection rate on dependency on the seed

## 8 Conclusion

Ant colony optimization algorithms and genetic algorithms are a convenient way to show satisfiability for boolean formulas. The algorithms presented in this paper are, however, unable to show that a formula is unsatisfiable. In this case, both algorithms will not terminate because they are run again over and over with different configuration profiles.

Both algorithms are metaheuristics. Therefore, they can easily be applied to different problems. Only the representation of solution candidates and basic operations like crossover and fitness evaluation need to be changed.

Besides ant colony optimization and genetic algorithms, other metaheuristics exist that we did not cover in this paper. For instance, tabu search is a simple algorithm that maintains a list of recently seen solution candidates or elementary changes that led to them. Another example is simulated annealing which is not a population-based algorithm and maintains only one solution candidate at a time.

Further research needs to done to find out how different algorithms perform in comparison to each other. Some concepts can probably be combined as we did with our concept of blurring pheromones in the ant colony optimization algorithm.

## References


1. Hans Kleine Büning. A study of evolutionary algorithms for the satisfiability problem. 2004.
2. Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.
3. Oscar Pérez Cruz and Alfredo Cruz. Evolutionary sat solver (ess).
4. Charles Darwin. *On the origin of species.* John Murray, 1850.
5. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.
6. M. Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992.
7. Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 1997.
8. John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.
9. Kenneth A. De Jong, Kenneth A. De, Jong William, and William M. Spears. Using genetic algorithms to solve np-complete problems, 1989.
10. Jin kao Hao and Raphaël Dorne. A new population-based method for satisfiability problems. In *Proceedings of the ECAI Workshop on Applied Genetic and other Evolutionary Algorithms*, pages 135–139. John Wiley & Sons, 1994.
11. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
12. R. Sethuram and M. Parashar. Ant colony optimization and its application to boolean satisfiability for digital vlsi circuits. In IEEE Computer Society Press, editor, *Proceedings of 14th International Conference on Advanced Computing and Communication (ADCOM 2006)*, pages 507–512, Mangalore, India, December 2006.
13. Thomas Stützle and Holger H. Hoos. Max-min ant system. *Future Gener. Comput. Syst.*, 16(9):889–914, June 2000.
14. Marcos Villagra and Benjamín Barán. Ant colony optimization with adaptive fitness function for satisfiability testing. In *Proceedings of the 14th international conference on Logic, language, information and computation*, WoLLIC'07, pages 352–361, Berlin, Heidelberg, 2007. Springer-Verlag.

# A Appendix

| name | type | #variables | #clauses |
|---|---|---|---|
| random_ksat1.dimacs | 5-CNF | 20 | 20 |
| random_ksat3.dimacs | 3-CNF | 100 | 300 |
| random_ksat8.dimacs | 3-CNF | 20 | 91 |
| random_ksat10.dimacs | 3-CNF | 50 | 216 |
| random_ksat11.dimacs | 3-CNF | 50 | 200 |
| random_ksat12.dimacs | 3-CNF | 50 | 210 |
| random_ksat13.dimacs | 3-CNF | 50 | 200 |
| random_ksat14.dimacs | 3-CNF | 50 | 216 |
| factoring21.dimacs | variable CNF | 369 | 83 |
| factoring22.dimacs | variable CNF | 369 | 83 |

**Fig. 17.** List of problem instances.

| **algorithm** | ant | | | | genetic | | | |
|---|---|---|---|---|---|---|---|---|
| **processes** | 1 | | 8 | | 1 | | 8 | |
| **factorings** | $f = 0$ | $f = 2$ | $f = 0$ | $f = 2$ | $f = 0$ | $f = 2$ | $f = 0$ | $f = 2$ |
| **name** | | | | | | | | |
| random_ksat1.dimacs | 0.110 | 0.118 | 0.109 | 0.116 | 0.130 | 0.117 | 0.116 | 0.121 |
| random_ksat3.dimacs | 0.631 | 0.714 | 0.523 | 0.714 | 0.827 | 1.091 | 1.370 | 1.607 |
| random_ksat8.dimacs | 0.211 | 0.139 | 0.120 | 0.135 | 0.246 | 0.145 | 0.147 | 0.140 |
| random_ksat10.dimacs | 3.047 | 1.881 | 4.101 | 3.517 | ∅ | ∅ | 13.369 | ∅ |
| random_ksat11.dimacs | 4.140 | 1.499 | 4.589 | 2.775 | 12.506 | ∅ | 0.593 | ∅ |
| random_ksat12.dimacs | 0.300 | 0.664 | 0.275 | 0.811 | 1.346 | 0.790 | 0.890 | 1.445 |
| random_ksat13.dimacs | 4.524 | 0.315 | 0.467 | 0.478 | 2.907 | 33.242 | 1.256 | 66.883 |
| random_ksat14.dimacs | 6.657 | 9.599 | 1.533 | 6.412 | ∅ | ∅ | ∅ | ∅ |
| factoring21.dimacs | 3.454 | 119.874 | 6.766 | ∅ | ∅ | ∅ | ∅ | ∅ |
| factoring22.dimacs | 22.557 | 4.057 | 5.051 | 8.234 | ∅ | ∅ | ∅ | ∅ |

**Fig. 18.** Runtime in seconds for ant colony optimization algorithm and genentic algorith. ∅ means timeout, i.e.the algorithm took more than 180 seconds. It is remarkable that runs with 8 threads are sometimes faster, or a least only a little bit slower, than runs with just one thread or two factorings (which normallay generates 4 threads), although we ran the benchmarks on a computer with just 4 CPU cores.