Tackling the Polarity Initialization Problem in SAT Solving Using a Genetic Algorithm

Sabrine Saouli, Souheib Baarir, Claude Dutheillet

Tackling the polarity initialization problem in SAT solving using a genetic algorithm

Sabrine Saouli[1], Souheib Baarir[2], and Claude Dutheillet[1]

[1] Sorbonne Université, CNRS, LIP6, 75005, Paris, France
[2] EPITA, LRE

Abstract. The Boolean satisfiability problem holds a significant place in computer science, finding applications across various domains. This problem consists of looking for a truth assignment to a given Boolean formula that either validates it or proves its impossibility.
An indispensable element influencing the efficacy of tools designed for tackling this challenge, known as SAT solvers, is the choice of an appropriate *initialization strategy*. This strategy encompasses the assignment of initial values, or polarities, to the variables before starting the search process. A well-crafted initialization strategy has the capability to curtail the search space and minimize the number of conflicts and backtracks by ensuring that variables are assigned values that are likely to satisfy the formula from the outset.
This paper introduces an innovative initialization approach founded on genetic algorithms, which are evolutionary algorithms inspired by the principles of natural selection and reproduction. Our approach executes a genetic algorithm on the given formula, persisting until it discovers a satisfying assignment or meets predetermined termination criteria.
Subsequently, it furnishes the satisfying assignment in case of success; otherwise, it employs the best assignment (that satisfies the highest number of clauses) to initialize the variables' polarities for the SAT solver.

Keywords: Boolean satisfiability · CDCL SAT solver · Initialization problem · Genetic algorithms · Evolutionary algorithms · Metaheuristic.

1 Introduction

Boolean satisfiability is a decision problem that consists in determining whether a given Boolean formula is satisfiable (SAT) (*i.e.*, there exists an assignment of values to the Boolean variables that satisfies all the constraints) or unsatisfiable (UNSAT) (*i.e.*, there is no such assignment that satisfies all the constraints simultaneously). SAT solvers are used to determine the satisfiability of a formula and return SAT (with a model) when it is satisfiable and UNSAT when it is not.

One of the challenges in SAT solving is the *initialization problem*. It has been defined in [12] as follows: given a SAT formula ϕ, compute an initial order over the variables of the formula and values/polarities for them in order to reduce the Conflict Driven Clause Learning (CDCL) [26] solver's run-time.

The initialization problem arises in Boolean satisfiability because the solver's efficiency can be greatly affected by the initial assignment. An initial assignment that is close to a satisfying one can quickly lead the solver to a solution that satisfies all clauses while a distant one can slow down the solving process because the solver may need to make more deductions before finding a satisfying assignment or determining that none exists. Therefore, finding a good initialization strategy is critical to the performance of SAT solvers. Various methods have been developed to generate initial assignments for SAT solvers [8,10,12,19].

In this paper, we focus on one aspect of the initialization problem: the polarity initialization of the variables (we refer to this problem as IPP throughout the paper). IPP has been addressed by various heuristics, such as stochastic local search [29], or probabilistic methods, such as Bayesian Moment Matching (BMM) [12]. However, these methods have some limitations. Stochastic local search, for instance, may be trapped in local optima, as it only explores a single solution at a time and its immediate surroundings. Moreover, BMM may be computationally expensive, as it needs to update and sample from a distribution. On the other hand, the genetic algorithm can overcome these challenges. It can explore diverse regions of the search space by applying crossover and mutation operators to a population of candidate solutions (assignments). It can also exploit the quality and diversity of the population by using selection and elitism operators. Consequently, we introduce a new approach, Genetic Algorithm for SAT Polarity Initialization (GASPI), based on a genetic algorithm, that aims at tackling IPP. Our goal is to use a genetic algorithm to initialize the polarity of variables in a way that enhances the performance of SAT solvers.

We implemented this approach in three different state-of-the-art SAT solvers: Kissat-MAB [24], Glucose [2], and MapleCOMSPS [21]. We evaluated our technique on the benchmark of SAT Competition 2022 [4] and compared it to the state of the art using two metrics: the number of solved instances and the PAR2 score (*Penalty Algorithm Runtime 2*). The results showed that our technique improve the performance of all three solvers. We also compared our approach to another initialization technique: BMM (Bayesian Moment Matching). The results showed that GASPI outperformed BMM on most of the instances.

The rest of this paper is organized as follows: In section 2 we recall the basic definitions and the technical background related to SAT-solving and genetic algorithms. In section 3, we review the related work on initialization techniques and genetic algorithms in SAT-solving. In Section 4, we present our novel approach for initializing the truth values of the variables in SAT solving using a genetic algorithm. Section 5, describes our experimental setup and results. Finally, in Section 6, we conclude and suggest some directions for future work.

2 Basic definitions and technical background

2.1 Boolean Satisfiability

A Boolean satisfiability problem is often represented by a Boolean formula in a Conjunctive Normal Form (CNF) in order to be solved by SAT solvers. The CNF

form expresses the formula F as a conjunction of clauses, such that $F = C_1 \wedge C_2 \wedge \ldots \wedge C_n$, where $n \geq 1$ and each C_i is a clause. Each clause is a disjunction of literals, such that $C = l_1 \vee l_2 \vee \ldots \vee l_k$, where $k \geq 1$. A literal is defined as either a Boolean variable or its negation where Boolean variables are the building blocks of SAT-solving. They are either true or false and represent the truth value of logical statements.

For a CNF formula F, an assignment α refers to a function that maps each Boolean variable to a truth value: true or false ($\top$ or $\bot$). If the assignment maps all the variables of the formula F, it is said to be *complete*, otherwise, it is *incomplete* or *partial*.

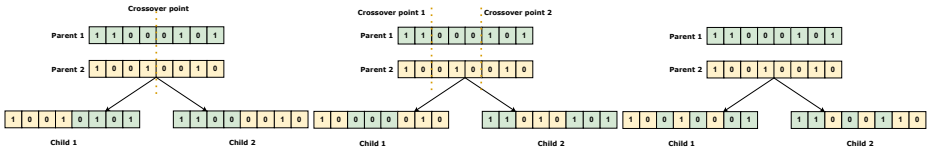
In SAT solving, the goal is to find at least one assignment that makes all the clauses in a CNF formula true. In this case, the satisfying assignment is called a *model* and the algorithm returns SAT. If no such assignment exists, the algorithm returns UNSAT. To achieve this goal, SAT solvers use a variety of algorithms and techniques. There are two main classes of algorithms for solving the satisfiability problem, namely *complete* and *incomplete* algorithms.

2.2 Complete and incomplete algorithms

Complete algorithms in SAT solving are those that guarantee a solution to the Boolean satisfiability problem, given enough time and memory. The most popular complete algorithm is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [11], which uses backtracking and unit propagation to search for a model. Another popular algorithm is the Conflict Driven Clause Learning (CDCL) algorithm [26], which extends DPLL with a conflict analysis mechanism that allows it to learn from conflicts encountered during the search process. This enables CDCL to quickly prune large portions of the search space and find solutions more efficiently than DPLL alone.

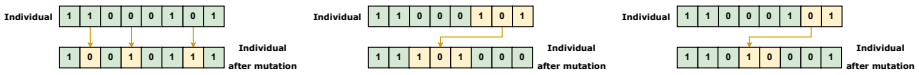
On the other hand, incomplete algorithms are based on a simpler approach known as stochastic local search (SLS) to solve the SAT problem. SLS algorithms rely on a series of heuristics and randomization techniques to guide the search toward a solution and require less memory compared to complete algorithms because they do not store any history of the previously explored assignments or learn any new clauses during the search. They only keep track of the current assignment and its fitness score and make local changes to improve it. The fitness here can be defined as a function that assigns a numerical value to each assignment based on its degree of compliance with the given formula, for example: the number of clauses satisfied by the assignment.

A typical SLS algorithm starts with a random initial assignment of variables and then iteratively modifies the assignment by randomly flipping the value of one variable. This is a simple and effective way to explore the search space, but it may also lead to stagnation. To overcome this problem, more recent SLS algorithms compute some scores for the variables to pick a variable to flip at each step. These scores are based on some heuristics that estimate the impact of flipping a variable on the overall fitness score of the assignment. For example, WalkSAT [25] algorithm uses the break-count heuristic, which counts the



(a) Single point crossover (b) Two points crossover (c) Uniform crossover

Fig. 1: Some crossover operators [27]



(a) Bit flip mutation (b) Displacement mutation (c) Simple inversion mutation

Fig. 2: Some mutation techniques [28]

number of clauses that become unsatisfied after flipping a variable. CCAnr [9] (Configuration Checking with Aspiration for Non-Random satisfiability) algorithm uses the configuration checking with aspiration heuristic, which measures the degree of satisfaction of each clause by an assignment. These heuristics help the SLS algorithms to avoid local optima and find better candidate solutions.

At each step, the SLS algorithms verify whether the new assignment satisfies the formula. If the current assignment is a model, the solver stops and returns SAT; otherwise, the process is repeated until a model is found or a termination condition is met. SLS solvers can be simple and fast since they do not exhaustively explore all possible combinations of variable assignments. However, they may not solve all problems, especially those that are UNSAT. Hence an incomplete algorithm returns SAT if it finds a model, otherwise, it returns UNKNOWN.

2.3 Genetic algorithms

Genetic algorithms [16] are powerful meta-heuristics that mimic the process of natural selection and evolution to seek optimal or near-optimal solutions for complex problems. They are based on the idea that a population of candidate solutions can be improved over generations by applying operators such as *crossover, mutation, and selection* that are inspired by biological mechanisms. Due to their versatility and effectiveness, genetic algorithms have found extensive applications in artificial intelligence, computing, engineering, and optimization domains.

A genetic algorithm (GA) follows a systematic process to evolve and improve a population of "candidate solutions" (for simplicity, these will be referred to as "solutions" in the rest of the paper).

It begins with a set of random solutions generated to form the initial population. Then, each solution in the population is assigned a fitness score that

measures how well it solves the problem. The fitness score is computed based on the objective function or constraints of the problem. During a *"Selection phase"*, a subset of solutions is chosen from the current population to produce offspring for the next generation. This process follows the principle of "survival of the fittest", favoring solutions with higher fitness scores. Various selection methods, such as roulette wheel, tournament, or rank-based, can be employed [18]. Afterward, a *"Crossover"* is performed by recombining pairs of selected solutions to create new offspring by exchanging parts of their encoding [27]. Inspired by biological chromosomal crossover (see Figure 1), this step introduces genetic diversity and allows the exploration of new regions in the search space. A *"Mutation phase"* follows, where each solution in the offspring population undergoes a small random change in one or more parts of its encoding [28] (see Figure 2). The crossover and mutation rates are parameters to be set. Then a subset of the solutions is selected to form the next generation. Different replacement strategies can be employed. For example, the entire current population can be replaced or only the worst solutions can be removed. The GA continues iterating through these previous phases until a stopping condition is met. Stopping conditions can be defined by a maximum number of generations, a minimum fitness score, a convergence criterion, or a combination of these.

3 Related works

3.1 IPP methods and heuristics

Various techniques and heuristics have been explored in previous research for initializing the truth values of the variables in SAT solving. One of them is *default initialization*, which is used by most of the modern SAT solvers. This technique simply sets the polarity of all variables to *false*. Some examples of SAT solvers that use this technique are MiniSAT [13], Glucose [2], and MapleCOMSPS [21].

Another technique is to use an SLS SAT solver as a preprocessor. This technique is used in KISSAT-like[3] SAT solvers [24] and consists in running an SLS solver on the SAT problem at hand for a limited amount of time to either solve it or simply use the best complete assignment found to initialize the variables' polarity. This initialization strategy enhances the search effectiveness of CDCL solvers. It is adopted by some of the top-performing SAT solvers in the latest competitions [3,4,14].

Another sophisticated heuristic-based method that was explored is the online Bayesian Moment Matching (BMM) heuristic [12]. This technique is implemented as a preprocessor that runs before the solver. It uses Bayesian inference to estimate the probability of each variable being true or false in a satisfiable formula. It then uses this information to initialize the values of the variables in a CDCL SAT solver. This technique was evaluated on a benchmark of real-world instances from various domains. It showed that it could improve the solver's performance in terms of the number of solved instances and average run-time.

[3] KISSAT is a CDCL SAT solver originally developed by A. Biere [6] and subsequently improved over time by many others, giving rise to a family of KISSAT-like solvers.

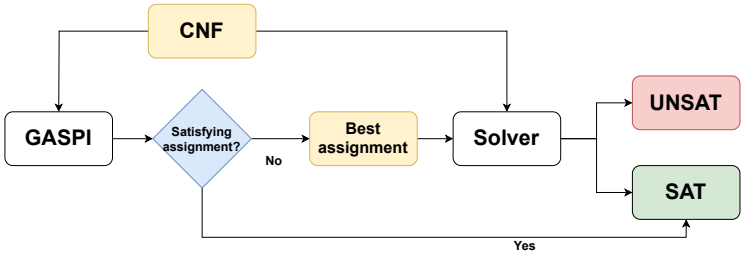


Fig. 3: Flowchart of GASPI

3.2 Genetic algorithms in SAT solving

GAs are a type of evolutionary algorithm that can be applied to various optimization and search problems, including the SAT problem [20]. Many studies have investigated the use of GAs for solving SAT problems, especially 3-SAT problems [1,22], and proposed different variations and enhancements of the basic algorithm. For instance, some methods have introduced different crossover and mutation techniques [5], hybridized GAs with unit propagation [7], or incorporated greedy strategy and effective restart [15] to improve the performance and accuracy of GAs for SAT solving.

Despite the various enhancements of GAs for SAT solving, they still face some challenges, such as slow convergence rate, suboptimal solution quality, and parameter tuning [23,28] and have not been able to compete with the state-of-the-art CDCL solvers.

However, GAs can still be useful for SAT solving as a preprocessing tool for CDCL solvers. They can explore a larger portion of the search space than SLS, which tends to get trapped in local optima. By using GAs to initialize the polarity of the problem's variables, we can provide a good starting point for CDCL. This preprocessing approach can improve the efficiency and accuracy of CDCL and enhance its performance, without relying on GAs to find the optimal solution.

4 Tackling IPP with GA

In this section, we introduce the proposed approach for tackling the IPP, Genetic Algorithm for SAT Polarity Initialization (GASPI), which uses a GA to initialize the polarity of the variables for the SAT problem. We explain how GASPI works, how it interacts with the CDCL solver, and how it evaluates the quality of the assignments. We also describe the different genetic operators that we use and how they affect the performance and convergence of GASPI. Finally, we present the parameters configuration that we use and how we determined their values.

4.1 Description of Genetic Algorithm for SAT Polarity Initialization (gaspi)

GASPI is designed as a preprocessing algorithm that receives a SAT problem as input and attempts to find a model. If such an assignment is found, GASPI returns True; otherwise, the SAT solver starts with an initial assignment corresponding to the polarities of the variables for the best individual in the final generation (See Figure 3).

Algorithm 1 shows how our initialization heuristic works. It follows the classical structure of a GA and takes as inputs the following parameters: `cnf`, the CNF formula representing the original problem, `numGen`, the maximum number of generations; `popSize`, the size of the population; `mutRate`, the mutation rate; `crossRate`, the crossover rate; and `occVars`, the set of variables that appear most frequently in the clauses of the treated problem. `crossOp`, `parentSelOp`, and `nextGenSelOp` represent the methods used for crossover, parent's selection, and selection of the individuals of the next generation respectively. Finally, `fitness`, that represents the fitness function used to measure the quality of the solutions.

Input: cnf, $popSize$, $numGen$, $mutRate$, $crossRate$, $occVars$, $crossOp$, $parentSelOp$, $nextGenSelOp$, $fitness$
Output: Complete assignment
1 Generate a random initial population;
2 **for** $i \leftarrow 1$ *to* $numGen$ **do**
3 evaluate the fitness of each individual in the population;
4 **if** *model found* **then**
5 **return** model;
6 **end**
7 Select parents for crossover using $parentSelOp$;
8 Perform single-point crossover with probability $crossRate$ using $crossOp$ operator;
9 Perform mutation with probability $mutRate$ on $occVars$;
10 Evaluate fitness of new individuals;
11 Select individuals for the next generation using $nextGenSelOp$ operator;
12 **end**
13 **return** best individual in final population;

Algorithm 1: GASPI Pseudocode

We then give some hints on the different steps of the GA algorithm and get into more details for some key operations: representation of the solutions, fitness evaluation, and mutation application.

4.1.1 Pseudocode:

The algorithm performs the following steps:

- It creates a random initial population of `popSize` individuals, where each individual represents a possible assignment for the SAT problem (line 1);
- It, then, evaluates the fitness of each individual in the population and keeps track of the best solution found so far (line 3);
- When an individual satisfies the CNF formula, it returns it as a model and declares the formula SAT (lines $4-6$) ;
- If no model is found, it selects half of the population as parents for reproduction using `parentSelOp` selection method. It then performs `crossOp` crossover with probability `crossRate` on pairs of parents to generate offspring (lines $7-8$);
- At this point, it performs mutation with probability `mutRate` on `occVars` to introduce diversity in the population. Mutation randomly flips some bits in a binary vector. `occVars` are the variables that have a higher impact on the fitness of the solutions, as they affect more clauses (line 9);
- It then evaluates the fitness of the new individuals created by crossover and mutation (line 10).
- Finally, individuals of the next generation are selected according to `nextGenSelOp` selection method (line 11).

4.1.2 Encoding: Our approach uses the following encoding:

- **Individuals:** An individual is a candidate solution. It is represented by a binary vector (representing a complete assignment) where each bit corresponds to the polarity of a variable of the SAT problem.
- **Population:** The population at each generation is represented by a vector of individuals.

4.1.3 Fitness function: It is a crucial component of a GA, as it measures the quality of a candidate solution for a given problem. The fitness function used in this paper evaluates a candidate solution by counting the number of unsatisfied clauses under the given assignment. The lower the fitness value, the better the solution, as it means that more clauses are satisfied. The optimal fitness value is zero, which indicates that the solution is a model for the problem. Therefore, the fitness function guides the GA towards finding models or, at least, assignments that satisfy as many clauses as possible.

4.1.4 Mutation: It can help introduce diversity and avoid premature convergence in the search process. However, mutation can also disrupt good solutions and reduce the quality of the population. Therefore, it is important to find the right balance when applying mutation. One way to do this is to mutate a limited number of variables and focus on the most influential ones in the problem.

In our context, we have to look for the variables that have the higher impact on the fitness of the solutions. Of course, this is not a trivial task but we can have some intuition about such variables with respect to their occurrence in the

clauses of the treated problem. Indeed, we think that the variables that appear in the greatest number of clauses of the problem at hand are the more influential. Hence, by mutating these variables, we can explore more promising regions of the search space and increase the chances of finding satisfying assignments. However, we have to determine a threshold for the number of variables that we take into account. This is the aim of the `occVars` parameter introduced in Algorithm 1.

4.2 Configuration of GASPI parameters

Configuration parameters, including population size, maximum number of generations, mutation and/or crossover rates, and specification of the genetic operators, are critical in the context of genetic algorithms because they profoundly influence the performance, behavior, and convergence characteristics of the algorithm. Here, convergence refers to the process of attaining a model or reaching a stable state in which the fitness ceases to progress significantly. The choice of each operator is crucial as it directly impacts which solutions are propagated to subsequent generations. Each method has its advantages and trade-offs in terms of maintaining diversity, preserving promising solutions, and enabling convergence toward optimal or near-optimal solutions.

The population size determines the number of potential solutions (chromosomes) considered in each generation. A larger population size searches a wider area but requires more memory and time while a smaller population size speeds up convergence but risks local optima. Therefore, it is a balance between exploration and exploitation. The number of generations also influences the algorithm's efficiency and effectiveness. Insufficient generations may cause the algorithm to stop before finding the optimal solution while too many generations can be computationally expensive without significant improvement in results. The number of generations is also related to the population size: Over a large number of generations, a small population size may negatively impact the diversity of the individuals, *i.e.,* all individuals in the population become very similar implying a premature convergence to a local minimum. Therefore, they should be considered together when designing a GA for a given problem.

Crossover and mutation probabilities represent the likelihood of genetic operators being applied to chromosomes. Crossover serves to transfer advantageous genetic traits to the succeeding generation, whereas mutation plays a vital role in introducing diversity within the population. For the crossover rate, the intuition is that a high value would mimic the living beings' nature of mating and reproduce novel and diverse solutions. So, we decided to fix its value to 0.95.

To preserve good genes in the next generation, we could use the elitist technique that selects the individuals of the current population with the best fitness values. This can improve the performance and convergence speed of the GA by preventing the loss of good solutions. However, elitist selection alone may also reduce the diversity of the population and cause premature convergence to a local optimum. To avoid this problem, we suggest introducing some randomness and diversity into the next population that will help the GA explore more regions of the search space and escape from local optima. The selection method for the

next generation would be a combination of elitist and random selection where half of the best-fit individuals are selected for the next generation, and the other half is selected randomly from the remaining individuals

Starting from a blend of classical parameter settings, reflecting the state-of-the-art, and our insights tailored to the specific needs of our SAT problem, we fixed the definition domains of the different parameters as follows: `popSize` ∈ {`20, 30`}, `numGen` ∈ {`10, 20, 30, 40, 50`}, `mutRate` ∈ {0.25, 0.50, 0.75, 0.88, 1}, `crossRate` ∈ {0.95}, `occVars` ∈ {10%}. `parentSelOp,nextGenSelOp` ∈ {Elitist, Random, Tournament, Elitist+Random}, `crossOp` ∈ {1-point, 2-points, 3-points}. As we can observe, the total number of possible configurations is 600, and testing all of them requires an excessively long computation time. Besides, there is no guarantee that the optimal values for one problem will be optimal for another. We decided then to select 5% of all these configurations that we thought promising and run a set of experiments using a random subset of 100 instances from the main track of the SAT Competition 2022 benchmark [4]. We conducted our experiments using a GASPI implementation, built upon the GLUCOSE solver. The results of our investigation revealed that specific parameter values had a notably more beneficial impact on our approach than others.

Accordingly, the parameter values we kept for the remaining experiments are as follows: `popSize:20`, `numGen:40`, `crossRate:0.95`, `mutRate:0.88`, `occVars:10% of the variables`, `parentSelOp:Random`, `nextGenSelOp:Elitist+Random`, `crossOp:1-point`.

5 Experimental results

In this section, we report the results of our experiments to evaluate the performance of our proposed approach, GASPI, when compared with different existing initialization methods (default initialization, SLS, and BMM).

All experiments were conducted on a computer with an Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz and 1500GB of memory. Each solver was run on each instance with a timeout of 7200s (including the polarity initialization time if used).

5.1 Solvers

To evaluate the performance of our approach, we implemented GASPI on three different baseline solvers: KISSAT-MAB [24], GLUCOSE [2], and MAPLECOMSPS [21]. These solvers are among the top performers in the SAT competitions in the last decade and use various techniques to improve their efficiency and effectiveness. We evaluated the performance of GASPI against three existing initialization methods: (1) the default initialization that assigns false to all Boolean variables, as used by GLUCOSE and MAPLECOMSPS; (2) the stochastic local search (SLS) initialization as used by KISSAT-MAB; and (3) the Bayesian Moment Matching (BMM) initialization that estimates the probabilities of the variables and assigns them accordingly, as implemented on GLUCOSE and MAPLECOMSPS. We name the different versions of the solvers we will evaluate as follows:

1. SLS-KISSAT: This is KISSAT-MAB that uses stochastic local search (SLS) initialization.
2. GLUCOSE: This is the base solver GLUCOSE-Syrup that uses the default initialization.
3. BMM-GLUCOSE: This is GLUCOSE with BMM initialization.
4. MAPLECOMSPS: This is the base solver MAPLECOMSPS that uses the default initialization.
5. BMM-MAPLECOMSPS: This is the base solver MAPLECOMSPS with BMM initialization.
6. GASPI-SOLVER: This is a solver that uses GASPI for polarity initialization (instead of its original initialization technique) such as GASPI-GLUCOSE [4], GASPI-KISSAT [5] or GASPI-MAPLECOMSPS [6].

5.2 Benchmarks

To evaluate the performance of our approach, we randomly selected 350 instances (due to time constraints) from the main track of the SAT competition 2022 benchmark [4], which covers different categories and difficulty levels. However, we believe that this sample size is still large enough to provide meaningful results. These selected instances can be found here https://zenodo.org/records/10819491

5.3 Results

The experiments were run three times for both SLS-KISSAT and GASPI-KISSAT on each instance because of the non-deterministic and random nature of the GA and the SLS. Both GA and SLS generate the first population randomly which can lead to different results in each run. Therefore, running the experiments multiple times can reduce the effect of randomness and provide a more reliable and robust evaluation.

Table 1 summarizes the results of our experiment.

The sub-tables 1a, 1b, and 1c report the number of solved instances when we consider the minimum, maximum, and average run time for each instance, respectively.

The minimum and maximum run times can be intuitive and easy to understand, but they may not reflect the overall performance of the configuration. Therefore, we also use the average run time, which is computed as the mean run time of the three runs for each instance, where we consider an instance as solved if the average run time is less than 7200s, and as unsolved otherwise.

The first column in each sub-table shows the name of the solver. The second, third, and fourth columns show the number of UNSAT, SAT, and total instances solved by each solver, respectively. Finally, the fifth column represents the PAR-2 score of each solver, which is a metric, used in SAT competitions, that evaluates

[4] https://github.com/sabrinesaouli/GASPIGLUCOSE
[5] https://github.com/sabrinesaouli/GASPIKISSAT
[6] https://github.com/sabrinesaouli/GASPIMAPLE

Solvers	UNSAT	SAT	Total (350)	PAR-2
VBS	129	141	270	1362534
SLS-KISSAT	127	133	260	1525795
GASPI-KISSAT	129	139	268	1405100

(a) Best run time

Solvers	UNSAT	SAT	Total (350)	PAR-2
VBS	126	126	252	1659595
SLS-KISSAT	124	122	246	1752335
GASPI-KISSAT	123	118	241	1816642

(b) Worst run time

Solvers	UNSAT	SAT	Total (350)	PAR-2
VBS	129	141	270	1453942
SLS-KISSAT	127	133	260	1594273
GASPI-KISSAT	129	139	268	1514151

(c) Average run time

Table 1: Evaluation of GASPI-KISSAT and SLS-KISSAT

the effectiveness of the solver by penalizing timeouts and errors. PAR-2 stands for *Penalty Algorithm Runtime 2*, and it is calculated by multiplying the run time of each instance by a factor of 2 if the solver failed to solve it or reported an incorrect answer. The lower the PAR-2 score, the better the solver.

When we consider the best and average results, GASPI-KISSAT solves more instances than SLS-KISSAT (8 more instances). Moreover, GASPI shows higher effectiveness on SAT instances than on UNSAT instances, as it solves only 2 more UNSAT instances and 6 more SAT instances than SLS-KISSAT in both the best and average cases. This suggests that GASPI can find models more reliably than SLS, which may get trapped in local optima. These results were in line with our expectations, as GASPI is designed to find a satisfiable assignment for the SAT problem. When the problem is unsatisfiable, GASPI may waste time and resources trying to find a solution that does not exist, or it may reach a local optimum that satisfies most but not all of the clauses. Therefore, GASPI has higher performance on SAT instances than on UNSAT instances. Furthermore, GASPI-KISSAT has a lower PAR-2 score than SLS-KISSAT, which means that it is more efficient and this can also be observed on the scatter plots of Fig 4.

Moreover, when evaluating the VBS (Virtual Best Solver) metric, which reflects the best performances combined, i.e. the number of instances that at least one solver can solve, we observe that it is always higher than the performance of each solver alone. In other words, GASPI-KISSAT solves instances that SLS-KISSAT didn't, and the other way around.

To go further in our study, we evaluated GASPI performance compared to the two other solvers: GLUCOSE and MAPLECOMSPS. We compared GASPI with the default initialization method of these solvers, which assigns false to all variables, as well as with the BMM initialization method. We used the same set of instances and ran GASPI-GLUCOSE and GASPI-MAPLECOMSPS only once (since the default initialization method and BMM initialization method are deterministic).

Table 2 presents the number of solved instances by GLUCOSE, MAPLECOMSPS, and their different initialization techniques. The results demonstrate that our approach improves the performance of both GLUCOSE and MAPLECOMSPS.

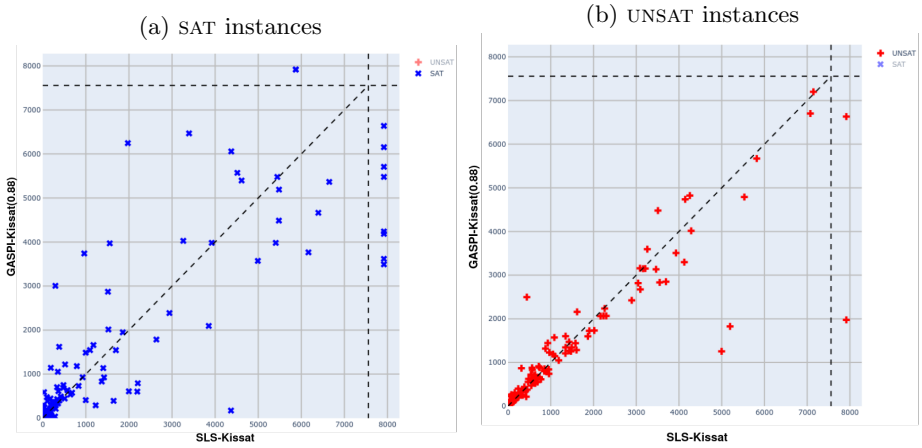


Fig. 4: Results of the comparison between SLS-KISSAT and GASPI-KISSAT(0.88) on the benchmark of 2022 when considering the average run time.

Solvers	UNSAT	SAT	Total (350)	PAR-2
GLUCOSE	99	72	171	2053133
BMM-GLUCOSE	99	75	174	2015878
GASPI-GLUCOSE	102	88	190	1920238
MAPLECOMSPS	107	101	208	2308575
BMM-MAPLECOMSPS	109	103	212	2295755
GASPI-MAPLECOMSPS	109	107	216	2259439

Table 2: Evaluation of GASPI-GLUCOSE and GASPI-MAPLECOMSPS

Even though the improvements are more significant in SAT instances, GASPI still outperforms the default and BMM initialization methods on UNSAT instances as well. These results are consistent with the previous observations.

6 Conclusion and future works

One of the challenges in Boolean satisfiability (SAT) is the initialization problem, which involves determining the optimal initial configuration for the variables in a SAT solver for a given problem. The nature of the problem influences these values, and the closer they are to a satisfying assignment (for satisfiable problems), the faster the SAT solver converges.

This paper proposes a novel approach, called GASPI, that employs a Genetic Algorithm as a preprocessor for the solving process. Starting from a CNF formula, it attempts to find an optimal or near-optimal complete assignment that satisfies all or most of the clauses. If no model is found after a predefined number of generations, GASPI assigns the preferred polarity of the variables based on the best values found. Then, the SAT solver starts and operates as usual.

The results show that our approach improves the performance of 3 different CDCL solvers, especially on satisfiable problems. When using GASPI for the initialization, the solver is able to find models for some instances that none of the other solvers could solve within the time limit. Our approach demonstrates the potential of using GA as a preprocessing technique for SAT solving, as it can exploit the structure and diversity of the problem domain.

Moreover, we observed that GASPI can generate diverse and effective initial assignments for the solvers. Therefore, one perspective would be to use this method as a diversification technique in parallel SAT solving, where each solver runs with a different initial assignment. This way, we can increase the chances of finding solutions for the solvers and cover more regions of the search space.

However, our approach has one limitation that we plan to address in future work: the difficulty of finding the right configuration for each problem. The parameters affect the quality and speed of the GA, and they may vary depending on the characteristics of the CNF formula. We are aware that there are some tools, such as SMAC [17] (Sequential Model-based Algorithm Configuration) that can help automatically tune the parameters of algorithms. Nevertheless, we decided to perform manual tuning as a first step because it can provide us with some insights into the behavior and sensitivity of our method concerning different parameter settings. Furthermore, manual tuning can serve as a baseline for comparing the performance of automatic tuning methods in the future. Therefore, another perspective would be to use machine learning techniques to automatically tune the GA parameters based on some features of the problem.

References

1. Aiman, U., Asrar, N.: Genetic algorithm based solution to sat-3 problem. Journal of Computer Sciences and Applications **3**(2), 33–39 (2015)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence. p. 399–404. IJCAI'09 (2009)
3. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M.: Proceedings of sat competition 2023: Solver, benchmark and proof checker descriptions (2023)
4. Balyo, T., Heule, M.J., Iser, M., Järvisalo, M., Suda, M.: Sat competition 2022 (2022)
5. Bhattacharjee, A., Chauhan, P.: Solving the sat problem using genetic algorithm. Adv. Sci. Technol. Eng. Syst **2**(4), 115–120 (2017)
6. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
7. Boughaci, D., Drias, H., Benhamou, B., et al.: Combining a unit propagation with genetic algorithms to solve max-sat problems (2008)
8. Braunstein, A., Mézard, M., Zecchina, R.: Survey propagation: An algorithm for satisfiability. Random Structures & Algorithms **27**(2), 201–226 (2005)
9. Cai, S., Luo, C., Su, K.: Ccanr: A configuration checking based local search solver for non-random satisfiability. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 1–8. Springer (2015)

10. Cai, S., Luo, C., Zhang, X., Zhang, J.: Improving local search for structured sat formulas via unit propagation based construct and cut initialization (short paper). In: 27th International Conference on Principles and Practice of Constraint Programming (CP 2021) (2021)
11. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**(7), 394–397 (1962)
12. Duan, H., Nejati, S., Trimponias, G., Poupart, P., Ganesh, V.: Online bayesian moment matching based sat solver heuristics. In: International Conference on Machine Learning. pp. 2710–2719. PMLR (2020)
13. Eén, N., Sörensson, N.: An extensible sat-solver. In: International Conference on Theory and Applications of Satisfiability Testing (2003)
14. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: Sat competition 2020. Artificial Intelligence **301**, 103572 (2021)
15. Fu, H., Xu, Y., Wu, G., Ning, X.: An improved genetic algorithm for solving 3-sat problems based on effective restart and greedy strategy. In: 2017 12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE). pp. 1–6. IEEE (2017)
16. Holland, J.H.: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT press (1992)
17. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5. pp. 507–523. Springer (2011)
18. Jebari, K., Madiafi, M., et al.: Selection methods for genetic algorithms. International Journal of Emerging Sciences **3**(4), 333–344 (2013)
19. Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Annals of mathematics and Artificial Intelligence **1**(1-4), 167–187 (1990)
20. Katoch, S., Chauhan, S.S., Kumar, V.: A review on genetic algorithm: past, present, and future. Multimedia tools and applications **80**, 8091–8126 (2021)
21. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: Maple-comsps, maplecomsps lrb, maplecomsps chb. Proceedings of SAT Competition **2016** (2016)
22. Marchiori, E., Rossi, C.: A flipping genetic algorithm for hard 3-sat problems (1999)
23. Rana, S., Heckendorn, R.B., Whitley, D.: A tractable walsh analysis of sat and its implications for genetic algorithms. Proceedings of the AAAI Conference on Artificial Intelligence **15**, 392–397 (1998)
24. Sami Cherif, M., Habet, D., Terrioux, C.: Un bandit manchot pour combiner CHB et VSIDS. In: Actes des 16èmes Journées Francophones de Programmation par Contraintes (JFPC). Nice, France (Jun 2021)
25. Selman, B., Kautz, H.A.: An empirical study of greedy local search for satisfiability testing. In: AAAI. vol. 93, pp. 46–51 (1993)
26. Silva, J.P.M., Sakallah, K.A.: Grasp—a new search algorithm for satisfiability. In: Proceedings of the 16th IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 220–227. IEEE (1997)
27. Soon, G.K., Guan, T.T., On, C.K., Alfred, R., Anthony, P.: A comparison on the performance of crossover techniques in video game. In: 2013 IEEE international conference on control system, computing and engineering. IEEE (2013)
28. Springer, P., Katoch, S.: A review on genetic algorithm: past, present, and future. Multimedia Tools and Applications **79**, 44651–44681 (2020)
29. Zhang, X., Cai, S., Chen, Z.: Improving cdcl via local search. Sat Comp 2021 p. 42 (2021)