An Enhanced Fuzzy-Genetic Algorithm to Solve Satisfiability Problems

José Francisco Saray Villamizar, Youakim Badr, Ajith Abraham
Institut National des Sciences Appliquées
INSA-Lyon, F-69621, France
{jose.saray, youakim.badr, ajith.abraham}@insa-lyon.fr

Abstract

The satisfiability is a decision problem that belongs to NP-complete class and has significant applications in various areas of computer science. Several works have proposed high-performance algorithms and solvers to explore the space of variables and look for satisfying assignments. Pedrycz, Succi and Shai [1] have studied a fuzzy-genetic approach which demonstrates that a formula of variables can be satisfiable by assigning Boolean variables to partial true values between 0 and 1. In this paper we improve this approach by proposing an improved fuzzy-genetic algorithm to avoid undesired convergence of variables to 0.5. The algorithm includes a repairing function that eliminates the recursion and maintains a reasonable computational convergence and adaptable population generation. Implementation and experimental results demonstrate the enhancement of solving satisfiability problems.

Keywords: Satisfiability, Genetic Algorithms, fuzzy logic, NP-Completeness, Evolutionary Computation

1. Introduction

The satisfiability is a decision problem to determine whether a Boolean formula only expressed in terms of AND, OR, NOT, variables, and parentheses can be entirely evaluate to true by assigning its variables to *false* and *true* values. Equally important is to understand that the function or the Boolean formula and its variables are all binary-valued. The question of determining whether a possible variable assignment exists is frequently referred to as a satisfiability problem [2]. The satisfiability problem (SAT) is of central importance in various areas of computer science including hardware design, computer-aided design, artificial intelligence, algorithms, electronic design, software verification, operations research and automation [1].

At first seen, the problem is an inoffensive. As the number of Boolean variables, N, becomes bigger, the search space grows in an exponential way to include 2^N possible combinations of variables which make the investigation to enumerate and find a solution a time-consuming [1]. Thus, any deterministic program fails to find a solution in a plausible time. The Cook–Levin theorem [3] proves that the Boolean satisfiability belongs to NP-complete class problems. It is the only problem in this class, because if any other problem exists, the cooke's theorem demonstrates that the problem can be reduced by a bijective function to a satisfiability problem. This is why SAT deserves a special interest in NP completness theorem: any NP-Complete problem can be solved by transforming it to a SAT instance [4].

Several works have studied the satisfiability problem and proposed high-performance heuristic algorithms and solvers to explore the space of variable assignments and look for satisfying assignments [5] [6] [7]. In particular, evolutionary and Genetic algorithms [8] have recently received a potential attention to find exact or approximate solutions to optimization and search problems. They reveal to be useful in solving satisfiability problems [13]. Pedrycz, Succi and Shai propose in their work [1] approach based on genetic algorithm and fuzzy logic theory to transform the binary domain of {0, 1} of Boolean variables to a continuous domain. After a random generating of fuzzy assignations they apply fuzzy logical operations to define the fitness function and solve a satisfiable problem of 200 variables. The major problem of this approach is the convergence of variable values to 0.5 in the case of more than 200 variables. It becomes difficult to safely decide whether the variable value should be truncated to 0 or 1. To this end, they develop recursive architecture to successfully avoid convergence to 0.5 and solve function of up to 1200 variables. The recursive-based version of the genetic

SAT solver still has some drawbacks because of its quadratic complexity and computational time consuming due crossover, mutation and selection operation.

In this paper, we improve the fuzzy genetic algorithm by introducing heuristic algorithm and repairing function which eliminate the recursion process and enhances drastically the computation time. The experimental results demonstrate the advantage of our approach with comparison to Pedrycz et al. algorithm.

The paper is organized as follows. In section 2 we present preliminary introduction to genetic approaches. The application of genetic algorithms and fuzzy theory are illustrated to formulate a satisfiability problem in the section 3. We introduce the repairing function and the pseudo code of our fuzzy-based genetic algorithm in section 4. In the section 5 we discuss the experimental results and show the enhancement of the algorithm. Finally we conclude the work and discuss future directives.

2. Overview of Genetic Algorithms

Genetic algorithms are a particular category of evolutionary algorithms which aim at finding exact or approximate solutions to optimization problems, they are encoded in binary strings and they use mutation and crossover for modifying the population through generations. In mathematics, the optimization problem seeks to minimize or maximize a function by choosing appropriate values for its variables. Genetic algorithms are inspired by biological evolution and based on genetic operations on genes, such as the mutation which changes the current value of a gene, and crossover that makes a new chromosome inheriting characteristics from the two parent chromosomes. These operations are applied together with Darwin's evolution theory, that states that individuals more fitted to the environment will survive, and will reproduce themselves maximizing their genetic code in the offspring, that will born with similar characteristics than their parents and by consequence they will be equally or better fitted to the same environment.

In the context of the evolution theory, individuals designate potential solutions to the optimized functions. Each solution is evaluated to check whether it minimizes or maximizes these functions in optimal way. Best fitted individuals are then selected [9] to generate new population through making crossover between individuals and applying mutation to the offspring to avoid premature convergence [8].

The genetic paradigm is a flexible approach enabling, for the same problem, different individual representations and algorithm implementations to select individuals and perform mutation. However, the appropriate representation of potential solutions is crucial to ensure that the mutation of any pair of individual (i.e. chromosome) will result in new valid and meaningful individual for the problem. Conversely, the choice of the fitness function that discriminates and converts back to Boolean space should be carefully studied [9].

3. Solving SAT Problems with Genetic Algorithms

As for the satisfiability problem, the potential solutions are commonly modeled in terms of binary strings each of which has a sequence of 0 or 1 [4], [6], [7]. However, the challenge becomes the selection of a suitable fitness function [14]. In the case that the function is the Boolean expression itself, its evaluation takes the truth value of 0 or 1 depending on all variable assignments. Boolean function does not help to provide appropriate fitness differentiation between individuals in our genetic optimization. A suitable solution could be a fitness function that makes the problem continuous so that each individual in the search space could come with a different value. The Fuzzy sets support the transformation from boolean space to continuous space. However, some other approaches are commented here:

De Jong and Spears [10] proposed AVERAGE function to replace AND operator and MAXIMUM function to replace OR operator, define the fitness function of the following expression as follows:

$F(x_1, x_2) = x_1 \, AND \, (x_1 \, OR \, \neg x_2)$

The fitness function will be:

$F(x) = AVERAGE \, (x_1, \text{MAXIMUM} \, (x_1, x_2))$

However, choosing such a function has several drawbacks because it does not preserve some of the important boolean laws:

Associativity law:
$(x_1 \text{ AND } x_2) \text{ AND } x_3 \equiv x_1 \text{ AND } (x_2 \text{ AND } x_3)$
Whereas
$AVERAGE \, (AVERAGE \, (x_1, x_2), x_3) \neq AVERAGE(x_1, AVERAGE \, (x_2, x_3))$

Morgan law:
$x_1 \, OR \, x_2 \equiv NOT \, ((NOT \, x_1) \, AND \, ((NOT \, x_2))$

whereas

MAXIMUM$(x_1, x_2) \neq 1 - ((1 - x_1) + (1 - x_2))/2$

Although the drawbacks of this fitness function, the algorithm in [10] succeeds to find a solution after 500 generations. The challenge of finding a suitable fitness function should solve the problem in a more efficient way. Alternative approaches can be found in the Fuzzy theory which considers a continuous logical domain instead of a binary domain. AND and OR operations are commonly replaced by a *triangular norm such as MIN* and *MAX* functions. In general, any mathematical operator that satisfies associativity, monotonicity and commutativity can be used for this purpose. The work in [1] proposed arithmetic *addition* and *product*, to overtake the fitness function drawbacks The approach assigns partial true values between 0 and 1 to different boolean variables composing the potential solutions and applying fitness function by using the arithmetic product norm. Upon completion of the algorithm execution the fitness fuzzy vector is converted to a binary vector by applying the following biasing function variable:

$$f(x_i) = 0 \quad \text{if } 0 \leq x_i < 0.5$$
$$f(x_i) = 1 \quad \text{if } 0.5 \leq x_i \leq 1$$

The disadvantage of using such a biasing function arises when the variable x_i is close to 0.5. The decision to convert x_i to 0 or 1 becomes non-deterministic.

The work in [1] demonstrates that the algorithm correctly solves Boolean expressions up to 200 variables with no undesired convergence to 0.5 after 40 generations and less than 2 hours of computation. After 200 generations, variables begin to inevitably converge to 0.5. The algorithm is then extended by transforming the genetic algorithm into a recursive function which succeeds to solve boolean expressions up to 1200 variables.

The major lack of this approach is due to the time-consuming and computational complexity. The fitness function should be applied to each potential solution, followed by a sequence of operations such as selection, crossover and finally mutation of chromosomes. The implementation of the recursive approach requires repeating several times computations that are already done. An interesting way to improve this approach consists of solving the undesired convergence to 0.5 by proposing an alternative solution to the recursion algorithm.

In the next section we introduce the repairing function to avoid the convergence to 0.5 and consequently reduce the computation complexity.

4. Enhanced Algorithm Avoiding 0.5 Convergence

The enhanced algorithm relies on the fact that AND operator is associative which means that Boolean formula can be rewritten in terms of AND operators and Boolean expressions. A potential solution for an AND operand implies the solution of the whole formula. This powerful property will enables the decomposition of a formula of N boolean variables to into P chunks (i.e. sub-expression) of M variables where M < N. The decomposition allows the genetic algorithm that does not find a solution for a given M-variable boolean expression to reapply the algorithm just for this M-variable expressions and not for the entire expression. In addition, the enhanced algorithm is extended with a repairing function to avoid 0.5 convergences. A repair function in genetic algorithms attempts to correct an individual (i.e. chromosomes) according to a particular problem to solve. For example, in [11] repairing functions are introduced to deal with scheduling constraints in the steel industry whereas in [2] they are introduced to deal with the TSP problems. In our context, the repairing function helps individuals to not converge to undesired value. The function guarantees that all variable assignments stay away from the value of 0.5.
The pseudo code for the enhanced genetic algorithm is illustrated below.

```
N:= NUMBER_OF_BOOLEAN_VARIABLES;
M:= CHUNK_LENGTH;
minterm := N_VARIABLE_MINTERM_TO_SATISFY;
number_of_chunks := N / M
remainder := N % M
p := randomly_initial_population(40, M );
indexini := 1;
indexfi := indexini + M;
index:= 1;
solution:=""
sol:="";

repeat
    repeat
        sol:= evolve(p,extrait(minterm,indexini,indexfi ));
    until sol = = extrait(minterm,indexini,indexfi) ;

    solution:=solution+sol ;
    indexini := indexini + M;
    indexfi := indexfi + M;
    index++;
until index ≤ number_of_times
indexfi := indexini + remainder;

repeat
   sol:= evolve(p,extrait(minterm,indexini,indexfi ));
until sol = = extrait(minterm,indexini,indexfi) ;
solution:=sol+evolve(p, extrait(minterm,indexini,indexfi))
```

The *evolve* function performs all necessary operations to transform generations based on the fitness function. The function *biasedSelection* randomly selects couple of individuals with highest fitness value whereas the function *makeCrossover* performs two chromosome's crossover. Furthermore, let *m* be the value that variables must not converge to and σ the distance from *m.* An algorithm for generating a random population is given by:

```
randomly_initial_population(POPULATION_SIZE: int,
CHUNK_SIZE:int)
    m:= 0.5;
    σ:=0.1;
    flag:=continue;
    population := Array[ 0 ... POPULATION_SIZE  ];
    chromosome := Array [0 ... CHUNK_SIZE ];
    n := POPULATION_SIZE;
    individual_index := 0
    repeat
            gene_index:= 1
              repeat
                  random_number = random (0, 1)
                  if ( m – σ ≤ random_number ≤ m + σ )
                      flag := continue
                  end if
                  else
                    flag:= stop
              until flag  == stop
               chromosome[gene_index] := random_number:
               gene_index:= gene_index + 1;
             until gene_index ≤ GENE_NUMBER
             population[individual_index] := chromosome;
             individual_index := individual_index + 1;
    until individual_index < n
```

Since the individuals in the initial population are distant from the undesired convergence value, the fuzzy-genetic algorithm can be applied. The algorithm implements crossover between variables which makes possible generate offspring convergence to 0.5. The repairing function, $f(x)$, helps to avoid the convergence. The function value of $f(x_i)$ substitutes the value of x_i in an offspring chromosome where $x_i \in [m - \sigma, m + \sigma]$. The repairing function defined as:

$$f(x_i) = m + \sigma \quad \text{if } m \leq x_i \leq m + \sigma$$
$$f(x_i) = m - \sigma \quad \text{if } m - \sigma \leq x < m$$

The repairing function improves the algorithm in [1]. In fact, the problem of 0.5 convergence is avoided because binary patterns that yield to value of 0.5 is completely extincted. In addition, the divide and conquer approach (i.e associative property) that allows the representation of the Boolean formula in terms of expressions and AND operators significantly reduces the number of generations and individuals implied in each computation.

```
evolve(population:array[0...NUMBER_CHROMOSOMES],fitnessFunction:S
tring): String)

return_chromosome := Array [0 ...fitnessFunction.size]

NUMBER_OF_GENERATIONS := 40;
ind_of_generations := 1;
ind_of crossover := 1;
repeat

  repeat
     chromo1 := biasedSelection (population);
     chromo2 := biasedSelection (population);
     makeCrossover( population, chromo1, chromo2);
     ind_of_crossover:= ind_of_crossover + 1;
   until ind_of_crossover ≤ NUMBER_CHROMOSOMES/2
    applyMutation(population);
   sortByFitness(fitnessFunction)
  return_chromosome:= population[NUMBER_CHROMOSOMES]
  ind_of_generations:=ind_of_generations + 1;
until ind_of_generations≤NUMBER_OF_GENERATIONS;

      return biasFunction(return_chromosome);
```

5. Experimental Results

We have developed a series of detailed experiments to demonstrate the feasibility of our algorithm and its repairing function. The algorithm was implemented using Java language and tested on a Pentium 4, 1.73 GHz and 1 GBytes RAM. All algorithm parameter values are selected by a trial and error strategy, until we got a performance in time and memory that reasonably improves the one proposed in [1].

Parameter	Value
Mutation rate	0.01
Number of chromosomes	40
Number of generations	40
Number of variables in chunk	14
Number of bits	32
Number of clones	6
σ - value	0.1
Crossover probability	0.85

Table 1. Parameter values

Same as the algorithm in [1], one single randomly generated minterm is chosen as Boolean expression to find the solution and reduce the computational time. Without loss of generality we can easily extend the solution to n minterms. As for the experimental results we implement the algorithm to solve a given minterm of N variables divided to P chunk-minterms each of which containing 14 variables. In addition, individuals are represented by vectors of decimal values between 0 and 1, each decimal component of the vector is encoded by 32 bit binary representation of the product between the decimal number and 2^{32}. A population consists of 40 individuals. An average of 40 generations was necessary to the algorithm to find a

appropriate solution to a chunk. The fitness function is calculated from the product norm of the fuzzy chromosome genes ($1-x_i$) or x_i depending whether the boolean variable of the index i in the minterm is complemented or not. We adopt a ceiling function to selection parents similar to the function proposed in [1]. The choice of the ceiling function favorites the individual selection as best fitness individuals. Each individual has a probability of crossover of 0.85. The crossover is made component by component, rather than once per pair of chromosomes. A mutation rate of 0.01 and six fittest individuals are cloned and included in the next generation. A σ value of 0.1 is used to guarantee that genes will always have a distance of 0.1 away from 0.5. A summary of these parameters is shown in table 1. Figures 1 and 2 illustrate the behavior of the fittest individual per generation and the average fitness for an expression of 20 variables and σ value of 0.1. As we can observe, the repairing function demonstrates that after 45 generations each gene is far enough from 0.5 where the fitness and average values begin to quickly increase.

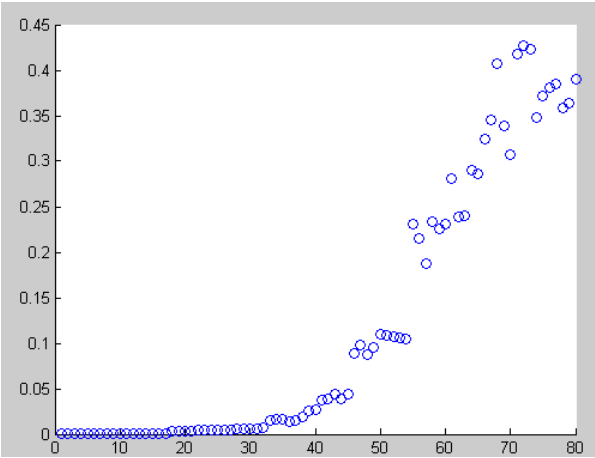

Figure 1. Best fitness individual per generation

Figure 3 shows a snapshot of the best fittest chromosome through generations. In contrast with the snapshots provided in [1], we observe that no Boolean variable is converging to 0.5. We can also check the result of the repair function. Some variables fluctuate around 0.4 and 0.6 values in the 20 generation and afterward they immediately begin to converge and get the desired solution after an average of 60 generation.

When the number of variables reaches the 200 threshold, some of variable values inevitably converge to 0.5 in [1] whereas applying the repairing function and the decomposition of N-variable functions into P groups of M variables, with M < N, makes possible to avoid the undesired convergence.

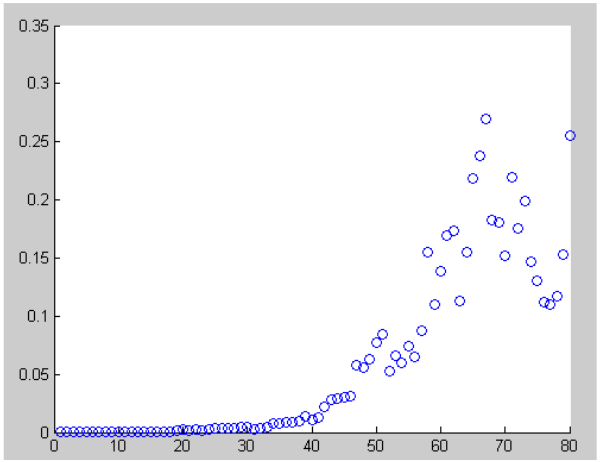

Figure 2. Average Fitness per generation

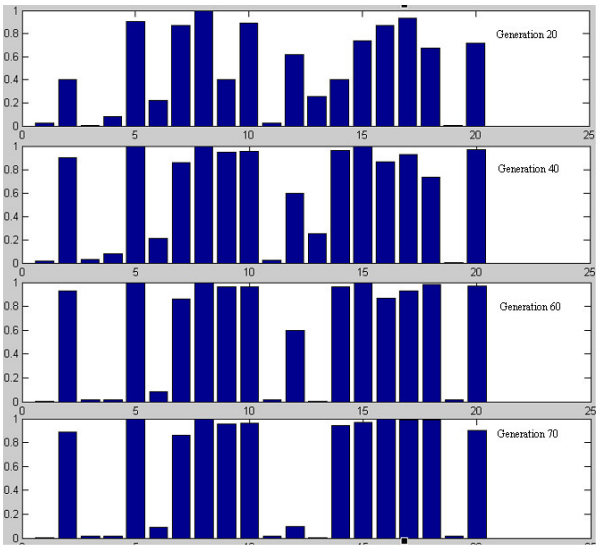

Figure 3. Snapshots of the fittest chromosome through generations

In addition, the time performance of the fuzzy genetic algorithm is improved since individual and generation numbers remain constant even if the number of variables increases (see Figure 4.). In contrast with the algorithm in [1], size and number of individuals and generations increase when the number of variables increases. Thus, the complexity of our algorithm is linear (correlation_coef(x,y) = 0.98456001) in contrast with the algorithm in [1] where the complexity is quadratic.

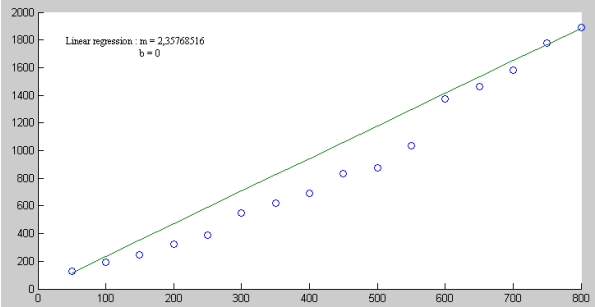

Figure 4. Correlation between the number of variables and algorithm execution time

The algorithm in [1] took 3 hours to solve 800 variable minterms, while our approach takes one hour and a half to solve 2400 variable minterms. Table 2 illustrates the significant improvement of the algorithm in term of computation and execution time.

Number of variables	Population size	Number of generations	Number of repetitions	Total time
800	40	40	57	00:31:32
500	40	40	35	00:14:36
300	40	40	21	00:09:12
100	40	40	7	00:03:15

Table 2. Algorithm time to solve variables assignments

6. Conclusion

The satisfiability is a decision problem that belongs to NP-complete class problems. It deserves a special interest since any NP-Complete problem can be solved by transforming it to a satisfiability instance. Genetic algorithms have recently received a potential attention to explore the space of variable assignments find solutions to satisfiability problems. In this paper we studied the Pedrycz, Succi and Shai work [1] which is based on genetic algorithm and fuzzy logic theory. We improved the fuzzy genetic approach by applying a divide and conquer strategy to reformulate the Boolean expression. We introduced a repairing function to eliminate the recursion process and enhances drastically the computation time. The experimental results demonstrate advantages of our algorithm in term of complexity, convergence and computational time. The future work includes the recursive application of the divide and conquers strategy in order to study the impact on the computational complexity.

7. References

[1] W. Pedrycz, G. Succi and O. Shai. *Genetic – Fuzzy Approach to the Boolean satisfiability Problem*. IEEE Transactions of evolutionary computation, Vol 6. No 5, October 2002.

[2] M.R.Garey and D.S. Jhonson, *Computers and Intractability, a guide to the theory of NP Completeness*. San Francisco CA: Freeman 1979

[3] S.A. Cook. *The complexity of Theorem Proving Procedures*. University of Toronto.1971

[4] J. Gotlieb, E. Marchiori and C. Rossi, *Evolutionary Algorithms for the Satisfiability Problem*. Evolutionary Computation, MIT press, vol.10, Nr.1, pp. 35-50, 2002.

[5] The SATisfiability problem website, http://www.satlive.org/ [Last visited December 2008]

[6] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, in *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1997

[7] S.A. Cook and D.G. Mitchell, *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1997.

[8] D. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Reading MA Addison Wesley, 1989.

[9] Z. Michalewicz, *Genetic Algorithms +Data Structures = Evolution Programs*. Springer Verlag. Third edition 1998

[10] K A Jong, W M Spears, *Using Genetic Algorithms to solve NP-Complete problems.* George Mason University, 1989

[11] J Dorn and M Guirch. *Genetic Operators Based On Constraint Repair*. Proceedings of the ECAI'94 Workshop on Applied Genetic and other Evolutionary Algorithms, Vienna University of Technology. 1994

[12] M.R. Bonyadi, M.R. Azghadi and H.S. Hosseini. *Solving traveling salesman problem using combinatorial evolutionary algorithm*. 4th IFIP Conference on Artificial Intelligence Applications & Innovations, Athens, Greece, 17-19 September, 2007

[13] W. Pedrycz, Computational Intelligence: An Introduction (tutorial), IASTED Int. Conf. on Applied Modelling and Simulation, Banff, July 27-August 1, 1997

[14] K. A. De Jong and W. M. Spears, "Using genetic algorithms to solve NP-complete problems," in Proc. 3rd Int. Conf. Genetic Algorithms. San Mateo, CA: Morgan Kaufmann, 1989, pp. 124–132.