Raport Badawczy
Research Report

RB/57/2001

Genetic – fuzzy approach to the boolean satisfiability problem

Witold Pedrycz, Giancarlo Succi,
Ofer Shai

Instytut Badań Systemowych
Polska Akademia Nauk

Systems Research Institute
Polish Academy of Sciences

POLSKA AKADEMIA NAUK

Instytut Badań Systemowych

ul. Newelska 6

01-447 Warszawa

tel.: (+48) (22) 8373578

fax: (+48) (22) 8372772

Pracę zgłosił: prof. dr hab.J.Kacprzyk

Warszawa 2001

Genetic - Fuzzy Approach to the Boolean Satisfiability Problem

[1,2] **Witold Pedrycz**, Fellow, IEEE, [1]**Giancarlo Succi**, [1]**Ofer Shai**

[1]**Department of Electrical and Computer Engineering**
University of Alberta
Edmonton, Canada T6G 2G7
and
[2]**Systems Research Institute, Polish Academy of Sciences**
Warsaw, Poland

Abstract This study concerns with the Boolean satisfiability (SAT) problem and its solution in the setting of a hybrid computational intelligence environment of genetic and fuzzy computing. In this framework, fuzzy sets realize an embedding principle meaning that original two-valued (Boolean) functions under investigation are extended to their continuous counterparts resulting in the form of fuzzy (multivalued) functions. In the sequel, the satisfiability problem is reformulated for the fuzzy functions and solved using a genetic algorithm (GA). It is shown that a GA, especially its recursive version, is an efficient tool for handling multivariable SAT problems. Thorough experiments revealed that the recursive version of the GA can solve SAT problems with more than 1,000 variables.

Keywords embedding principle, fuzzy sets, fuzzy functions, Boolean functions, triangular norms, hybrid approach, computational intelligence

1. Introduction

The satisfiability problem (SAT) [4][6] [12] concerns a determination if a given Boolean function (f) of n variables is satisfied, meaning that there exists a combination of its arguments for which this function attains a logical one (we say it is *satisfied*). In other words, we are interested in determining logic (truth) values of the variables $x_1, x_2,\ldots, x_n$ for which $f(x_1, x_2,\ldots, x_n)$ achieves one, namely, $f(x_1, x_2, \ldots, x_n) = 1$. While the problem seems obvious, a solution is not trivial. Even for a modest number of variables, a brute force enumeration fails as we are faced with a profound combinatorial explosion. For a Boolean function of n variables, a straightforward enumeration requires an investigation of 2^n combinations of the inputs (Boolean variables). This number increases very quickly. To quantify this, let us assume that a single evaluation of the Boolean function takes 1μs ($=10^{-6}$s). For only $n = 50$ it would take 35 years to complete an exhaustive (brute-force) enumeration of all combinations and find a solution. The SAT problem is a classic example of a NP-complete problem [9] meaning that there is no known algorithm that solves it in polynomial time [12]. Put differently: a worst-case running time of a SAT solver grows exponentially with the number of variables. SAT is a fundamental problem in logic and computing theory. It has numerous applications to automated reasoning, databases, computer-aided design, and computer architectures [4][6][11], to name a few. The use of SAT to automatic test generation of patterns to test digital systems is an attractive application.

Owing to the immense size of the search space in the SAT problem, evolutionary computing arises as a viable and attractive option. The objective of this study is to formulate the SAT problem in the evolutionary setting and carry out comprehensive experimental studies. The approach relies on the embedding principle: we generalize the Boolean problem to its continuous fuzzy (multivalued) version, find a solution to it, and convert (decode) it to the Boolean format. The concept of this transformation (embedding) was introduced initially in [10]. This study follows by elaborating on the algorithm, presenting results of comprehensive experimentation, and discussing improvements to a generic genetic algorithm (GA) necessary in the case of high-dimensional SAT problems.

We confine discussion to the basic binary model of GA. The material is organized into 7 sections. First, we formulate the SAT problem in the GA environment by introducing an embedding principle that shows how a binary problem can be embedded into a continuous environment of fuzzy (multivalued) functions generated in the setting of fuzzy sets. Then we discuss details concerning the experimental environment (Section 3) including genetic optimization and a way of generating Boolean functions. In Section 4, we discuss experimental results, the efficiency of GA in solving the SAT problem and contrast this approach with random search and brute-force complete enumeration method. Moreover, we discuss an issue of scalability of the problem and experimentally identify some limits as to the number of Boolean variables. Afterwards, a recursive version of the genetic SAT solver is discussed in Section 5. It is shown how this recursive approach

helps to handle a high-dimensional problem. Conclusions are contained in Section 6. References are included in Section 7.

2. The SAT in an Evolutionary Environment

When formulating the SAT problem in the framework of evolutionary computing, we need to revisit the main algorithmic components of a GA and define them in a proper way according to the specifics of the problem at hand. Starting from the fitness function, we immediately encounter a significant conceptual and technical problem. As we are dealing with the Boolean functions, an immediate form of the fitness function that comes to mind would be the Boolean function itself. This definition, however, does not help at all. The combination of the variables producing the value of the Boolean function equal to 1 is just a solution to the problem. The zero value of the Boolean function means that these specific inputs are not a solution. All nonsolutions are the same from the standpoint of the fitness function. Evidently, as being indistinguishable they are not helpful for any genetic optimization. The drawback in the definition of the fitness function is implied inherently by the nature of the Boolean problem. The choice of the fitness function is quite challenging, as indicated in [2]. For instance, in [9] the Boolean variables are changed into floating-point numbers; in this way one tries to consider the solution to the SAT problem to correspond to a set of global minimum points of the induced objective function. None of these approaches have addressed an issue of retaining the logic character of the original problem.

The approach taken here it is to make the problem continuous so that each element in the search space could come with a different value of the fitness function. Moreover, to make the binary (Boolean) problem continuous and still maintain its logic character brings us into the world of fuzzy sets and fuzzy functions. Fuzzy sets support the embedding principle: instead of the original problem, we cast it into the format of the corresponding fuzzy function, solve the problem in this new framework, and bring back (transform) the solution to the original binary environment. This principle is illustrated in Figure 1.

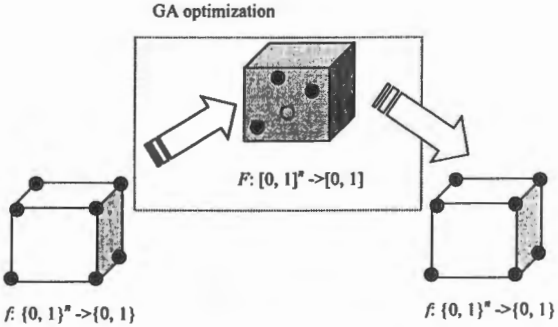


Figure 1. From binary logic to fuzzy logic: the essence of the embedding principle. The original binary world $\{0,1\}^n$ is made continuous $[0,1]^n$, genetic optimization is carried out there and the derived solution is converted back (decoded) to the Boolean format. Note that the binary world confines to the vertices of the unit hypercube whereas the world of the fuzzy functions involves any interior point of the unit hypercube

The fuzzy set extension of the problem subsumes the binary version. In other words, if $F(x_1, x_2, \ldots x_n)$ is a continuous (generalized) version of the induced Boolean problem (where x_k is in the unit interval), it immediately collapses to the original binary version once we confine ourselves to $x_k \in \{0,1\}$, namely

$$F(x_1, x_2, \ldots, x_n)_{|x_k \in [0,1]} = f(x_1, x_2, \ldots, x_n)_{|x_k \in \{0,1\}}$$

The fundamental fuzzy set operators such as triangular norms [10] help generate a fuzzy function f (its generalization) by allowing x_k to take any value in [0,1] and use any t-norm as a logic *and* and view any s-norms as a realization of the *or* operation. This makes the fuzzy function logically consistent with the binary version of the problem (Boolean function). For instance, if the original Boolean function reads as

$$f(x_1, x_2, x_3) = x_1 + x_2 x_3,$$

then its generalization to the continuous world arises as

$$F(x_1, x_2, x_3) = x_1 \; s \; (x_2 \; t \; x_3)$$

with t and s denoting a certain t- and s-norm, respectively. The choice of the triangular norms may affect the performance of the GA. For the Boolean case, all t- and s-norms become equivalent (and this is a consequence of the boundary properties of these operators, namely $at0 = 0$, $at1 = a$, $as0 = a$, $as1 = 1$ for any truth value a coming the unit interval). In the world of fuzzy logic, the same function realized via different t- and s-

norms may differ quite substantially and exhibit different levels of computational efficiency. The "classic" examples of triangular norms and co-norms are the minimum (t-norm) and maximum (s-norm) functions. They are easy to compute. Nevertheless they exhibit a lack of "interaction" that may become an evident drawback when navigating through the search space. Another quite common option of the logic connectives are the product (t-norm), $atb = ab$ and probabilistic sum ($asb = a+b- ab$). This particular pair of the t - and s-norm is a model of alternative connectives in Boolean logic encountered in Boolean logic, cf. [11] (originally they were introduced by G. Boole himself in his famous *The Laws of Thought* (1854) in the following form x OR $y = x + y(1-x) = x + y - xy$ x AND $y = xy$)

Fitness can be defined as equal to the value of the fuzzy function assumed for some given values of the arguments. As opposed to Boolean functions, fuzzy functions are satisfied to a certain degree. These satisfaction levels help discriminate between various elements in the search space and guide the evolutionary optimization process.

Once the optimization has been completed, the solution in the continuous space (space of fuzzy functions) has to be converted back (decoded) to the Boolean space. A simple threshold operation is a sound option

- *if* x_k is less than 0.5 *then* convert x_k to 0 otherwise convert x_k to 1

Intuitively, the closer the truth value of x_k to 0 or 1, the more confident we could be about the thresholding rule. If x_k gets closer to 0.5 (eventually being equal to 0.5), the more hesitation arises as to its conversion to 0 or 1 and an overall credibility of such processes. Further on, we show that this intuitive observation may be helpful in the development of a recursive architecture of the genetic SAT optimization.

3. The experimental setting

3. 1. Evolutionary optimization

The evolutionary optimization is realized in the form of a standard GA as commonly encountered in the literature, [1][3][5] [8]. The format of the problem implies a form of the genotype. As we are concerned with fuzzy functions and fuzzy (multivalued) variables, each variable is coded in a binary format. Each variable is coded in 32 bits representing a real value in the the range [0,1].

Once a population is generated, the individuals (chromosomes) are sorted according to their fitness. A procedure is then used to select two individuals to mate. The procedure for selecting the two individuals gives preference to higher fitness individuals. Two random numbers, in a range larger than the number of actual individuals in the population, are chosen and mapped into two individuals from the actual population. The mapping is done in a way that creates a funnel effect, where individuals with higher fitness have higher priority. A parameter governing the range of the two original numbers affects the width of the funnel, and controls the diversity in choosing the individuals: a

very strong funnel immensely discourages low-fitness individuals from mating and passing on to the next generation. The single-point crossover itself traverses each pair of solutions and crosses each variable in the first solution with the corresponding variable in the second one.

Each offspring, in turn, is subject to mutation with some probability (mutation probability). Once the individual is chosen, it is traversed, and again, each variable undergoes a single-entry mutation (i.e., for each 32-bit string-variable, a single bit is flipped).

The best individual in a population is then decoded to the binary format; if it satisfies the Boolean function, the process is complete, otherwise we proceed with the next generation. The fitness function evaluates the fuzzy value and the Boolean value of each individual together. The fitness value actually given to the individual is its fuzzy value, unless its Boolean value is 1, in which case 1 is assigned. Since all the other individuals have fitness values in (0,1), a solution automatically becomes the best individual. In the following generation, a few of the best individuals in a current population appear in the next population (elitist strategy). Throughout the series of experiments, we use some general parameters such as

- population size
- number of generations. There is the maximum number of generations the GA will run for but it stops once a solution is found
- number of clones for each generation. These are the elitist individuals.
- crossover parameter
- probability of mutation

Subsequently, some experimentation was carried out to explore the affect of these parameters on the performance of the algorithm. Other than very general trends, however, no specific impact was observed by changing one parameter. The general trends were exploited to determine the best parameters' values; those were used in later experiments.

3.2. Generation of Boolean functions

In most experiments, a Boolean function was realized as a single minterm (that is a product of all variables, coming either in direct or complemented form). In general, Boolean functions were realized in their minterm representation. Several strings were generated randomly, each representing a minterm of the function. The strings were composed of 1's and 0's where each 0 represented a complemented variable, and a 1 represented a non-complemented variable. Then the minterms were combined together through an OR operation. The use of these Boolean functions was time consuming. Since each individual's fitness was dependent on all minterms in the function, a five minterm function took roughly five times longer to evaluate than a single minterm function, and the respective GA behaves correspondingly. At the same time, any single minterm function forms the most challenging environment (as only one combination out of total of 2^n leads to the satisfaction of the function). Having this in mind, we decided to

experiment with single minterm Boolean functions. These Boolean functions would look rather simplistic to a human observer yet they are the most challenging from the optimization point of view. Obviously, assigning 1 to the noncomplemented variables and 0 to the complemented ones would provide the desired solution. This, however, is not evident to the program, since it went about solving it using a genetic algorithm rather than by direct observation as a human would. Using this simple representation for the function provided for a double advantage: the program's run time evaluation was fast, and verifying that the result the program provided was a matter of comparing two strings of numbers.

4. Experimental studies

We completed a series of detailed experiments. The performance of GA is reported in terms of the performance of the GA and detailed results (both for the fuzzy functions as well as Boolean functions). Our interest is also to explore the use of different realizations of the triangular norms. They behave in the same way for the binary case yet they may have significant impact on the performance of the genetic optimization. It is also of interest to investigate how well the GA approach scales up, that is, how well it performs when the size of the problem (number of variables) increases.

4.1. Experimental settings and results

The starting point is a rather small, oneminterm Boolean function with $n = 20$ variables. The parameters of the experiment are listed in Table 1.

Population size	200
Maximum number of generations	200
Number of clones	8
Crossover rate	0.4
Probability of mutation	0.1

Table 1. A list of parameters of the GA experiment: 20-variable Boolean function

Figure 2 and 3 summarize the performance of the GA in terms of the fitness function (both the best individual as well as an average for the entire population). The plot is a result of 20 experiments. In general, we found a high reproducibility of the overall behavior of the optimization scheme. Two pairs of triangular norms are investigated: (a) probabilistic sum (s-norm) defined as $asb = a + b - ab$ and product (t-norm), $atb = ab$, and (b) maximum (s-norm), $\max(a,b)$ and minimum (t-norm), $\min(a,b)$, $a, b \in [0,1]$.

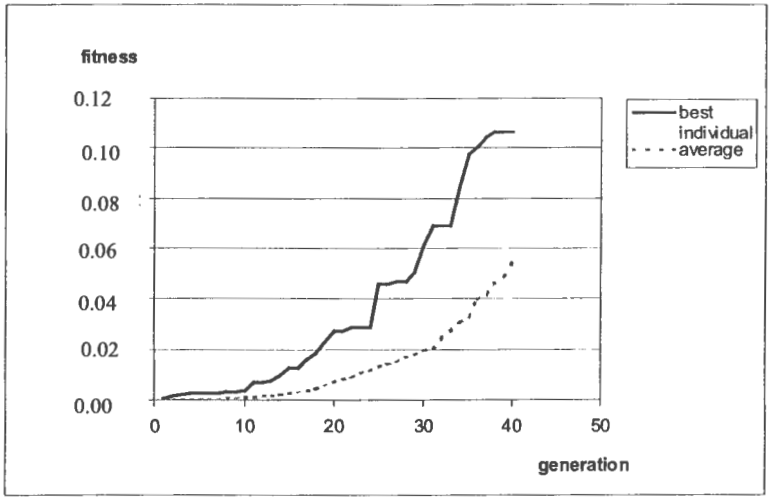


Figure 2. Fitness function in successive generations (triangular norms: probabilistic sum and product)

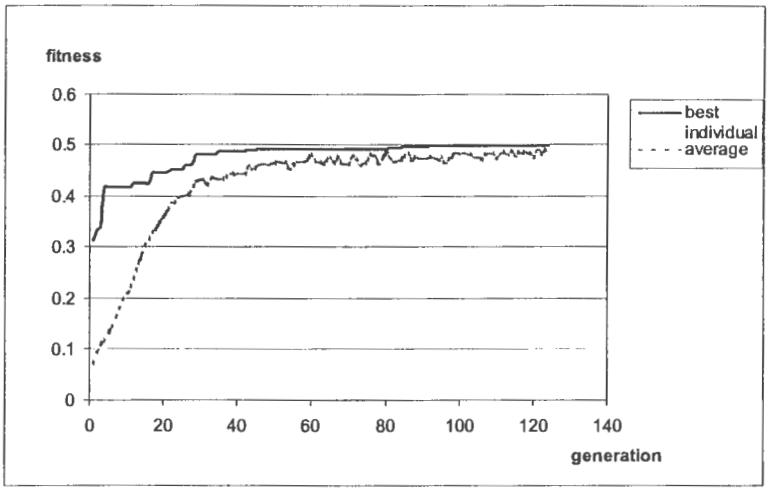
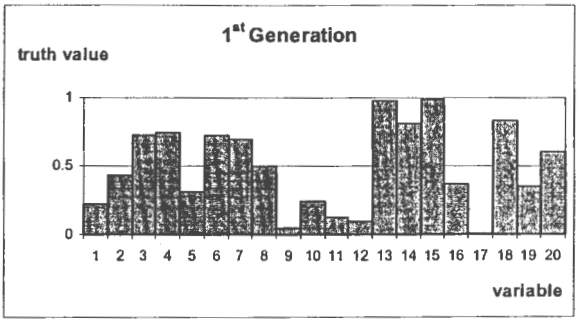
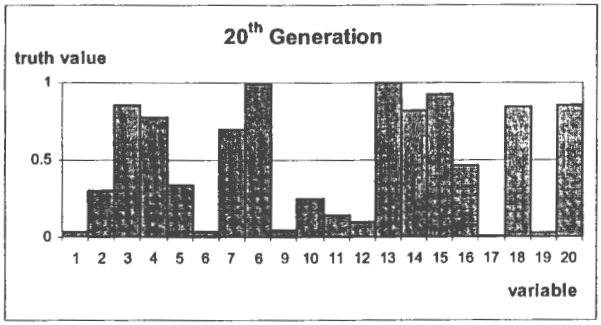


Figure 3. Fitness function in successive generations (triangular norms: maximum and minimum)

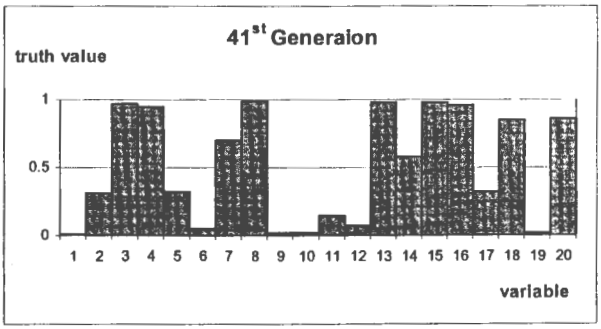
The comparison of the effectiveness of different *t*-norms and *s*-norms based exclusively on Figure 2 and 3 does not reveal a complete picture. In the first set of the triangular norms, the solution was achieved after 41 generations. In the second scenario, the SAT was accomplished after 125 generations. Figure 4 provides a better insight into the nature of the solution (GA produces a solution in the unit hypercube that has to be then decoded into a Boolean format) and the way we arrive at the solution during the optimization.



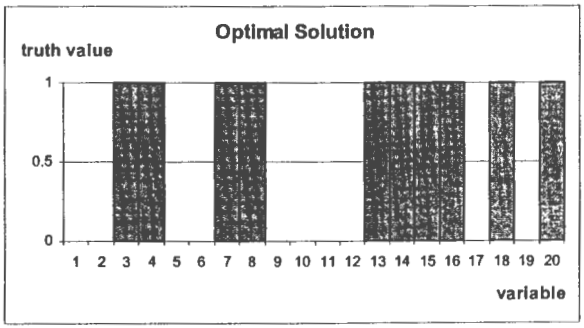
(a)


20th Generation
truth value
1
0.5
0
1 2 3 4 5 6 7 6 9 10 11 12 13 14 15 16 17 18 19 20
variable

(b)


41st Generaion
truth value
1
0.5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
variable

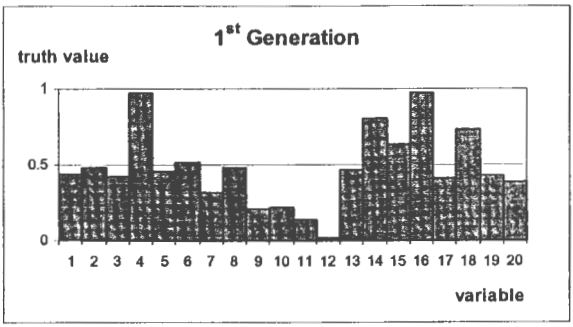
(c)


Optimal Solution
truth value
1
0.5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
variable

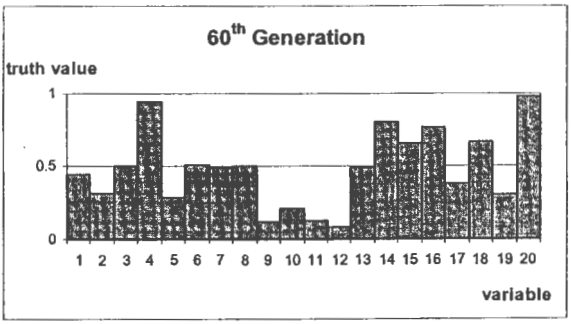
(d)

Figure 4. Snapshots of the GA solution in the [0,1] truth space for some selected generations: (a) 1st generation, (b) 25th generation (c) 48th generation (at which the SAT problem was solved) and (d) the single-minterm Boolean function used in the experiment; the triangular norms selected as probabilistic sum and product. The parameters are as described in table 1.

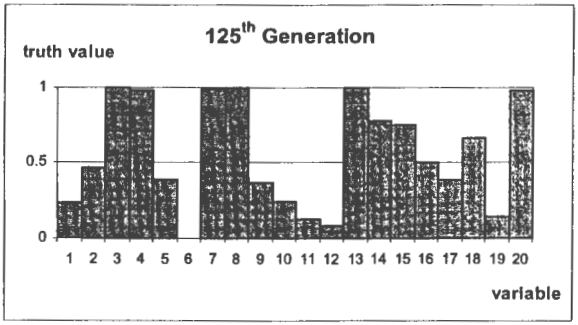
Figure 4 reveals an interesting pattern illustrating how a Boolean solution has been reached. At the beginning (in the first generation), there are a number of fuzzy variables assuming truth values around 0.5. In subsequent generations, the solution starts to emerge gradually: while in the 20th generation, we still encounter a number of "undecided" variables, they tend to vanish as clearly visible in Figure 4 (c) where the solution has been reached in the 41st generation.



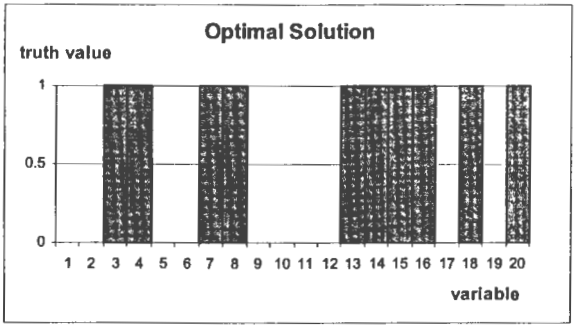
(a)



(b)



(c)



(d)

Figure 5. Snapshots of the GA solution in the [0,1] truth space for some selected generations: (a) 1st generation, (b) 50th generation (c) 116th generation (at which the SAT problem was solved) and (d) the single-minterm Boolean function used in the experiment; the triangular norms selected as minimum and maximum. The parameters are as described in Table 1.

Figure 5, in conjunction with Figure 3, shows the progression of the GA toward finding the solution for the minimum and maximum norms and reveals the reason for the poorer performance by those norms. Due to the non-interactive nature of these norms, the fitness value is determined solely by the variable that is farthest from its optimal value. This slows down the convergence of the other variables. In fact, where using the probabilistic sum and product norms, the fitness value was not an indication of how close the algorithm was to finding a solution, here the solution is found as soon as the fitness exceeds 0.5, since that indicates that all variables have crossed their respective thresholds.

4.2. Run-time analysis

The timing aspect of the SAT problem paints a very convincing picture. Here we contrast between three options of the SAT problem solving, that is (a) brute-force enumeration, (b) random search, and (c) the GA approach. The comparison is completed for the same hardware environment used in the previous experimentation that is Pentium III 450, 128Mb RAM, 8.0 GB hard disk (standard, no high-end dedicated hardware). The programming environment was C++, compiled using GNU g++ under Linux.

The results of optimization are summarized as follows:

- Brute-force (exhaustive) enumeration As underlined before, the method is viable for a very small number of Boolean variables and scales up very poorly due to the NP nature of the problem. For 20 variables, it takes about 3 seconds to complete the search. 25 variables require the search time in the range of 3 minutes and 30 variables took about 75 minutes to sweep through the search space.
- Randomly generating individuals and hoping to hit the solution alleviates the need to search through the entire search space. Each population size was given 20 chances to randomly generate individuals, such that the number of individuals generated is equal to the GA search space for the equivalent population size. For 20 variables, the random search was able to find solution 10% of the time (here we mean that 10% of experiments initiated randomly and run for this size of the problem was successful). This percentage goes down to 5% for 25 variables. The random search was not successful for higher numbers of variables.
- The genetic algorithm scales up quite nicely resulting in the search time of 1 sec for 30 variables, 75 secs for 100 variables, about 35 minutes for 150 variables and about 2 hours in the case of less than 200 variables. In all cases the solution was found. One should stress, however, that the this successful scaling effect was observed until to this number and then the method started to fail (which triggered our interest in a recursive format of the algorithm). Figure 6 summarizes run-time as a function of the number of variables used in the problem.

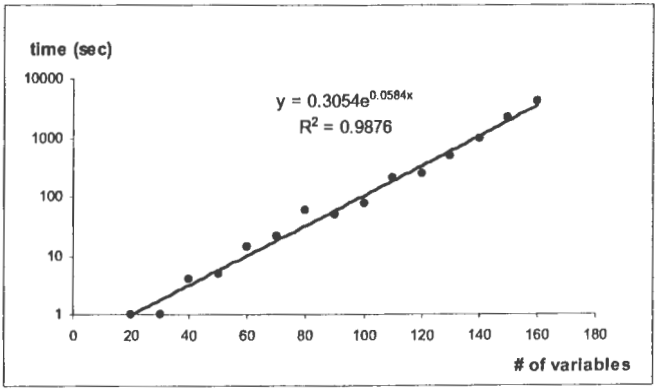


Figure 6. Run-time vs. the number of variables in the function

From the above analysis, it becomes apparent that the GA is a computationally viable and attractive option. In addition to the time complexity analysis, it is of interest to analyze how well the GA did with regard to the search space. The search space is enormous for higher values of n. The genetic search was successful after searching only a fraction of the entire space. We may define a space exploration rate that is a ratio of the number of all elements investigated by the GA and the total number of elements in the search space. The lower the ratio, the more efficient the GA is. The number of elements visited through the GA search is determined based on the size of the population used in the experiment and the number of the generations. As shown in Table 2, this ratio is extremely low.

Number of Boolean variables (n)	20	50	80	110	140
Space exploration rate	1.32%	$6.13*10^{-11}$%	$4.407*10^{-17}$%	$1.07*10^{-25}$%	$2.35*10^{-34}$%

Table 2. Space exploration rate for different sizes of the SAT problem

4.3. Boolean function with many minterms

The previous experiments were concerned with single minterm Boolean functions (let us again stress here that this type of function is not an impediment but forms the most challenging environment for the SAT). Note that with the increase of the number of minterms, the chances of solving the problem go up. In this sense, the experiments discussed in Section 4.1 were the most demanding). Nevertheless from the experimental point of view it is of interest to look into the performance of the GA approach for Boolean functions with many minterms. In the current experiment, we shows the increase in time as the number of variables grow. There are 0.3 * variables minterms (i.e., for 20 variables, 6 minterms, for 50 variables, 15 minterms, etc.). Figure 7 shows the time required to solve the SAT problem when dealing with the variable size of the problem.

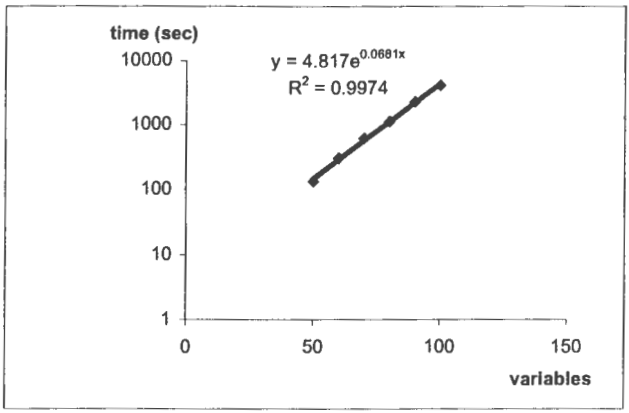


Figure 7. Run-time versus the number of Boolean variables

In Figure 8 we visualize how the fitness function changes over optimization. In this particular case we have selected a 100 variable function with 29 minterms, that took time very close to the average. Contrasting this relationship with the previous findings, we note that it is quite close in a way in which the fitness function changes throughout generations.

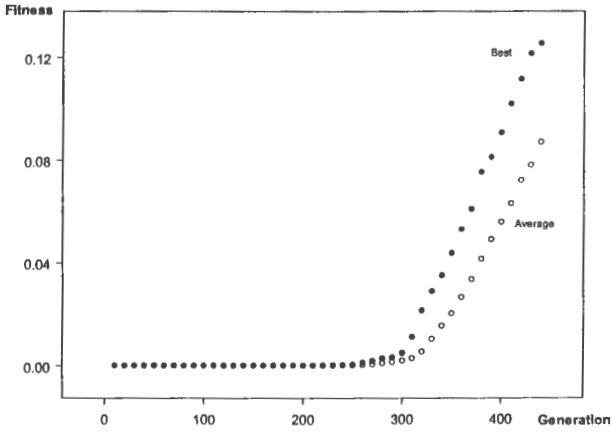
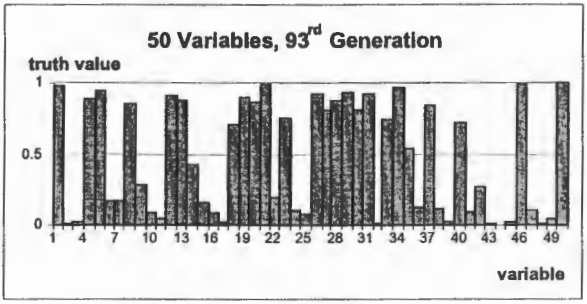


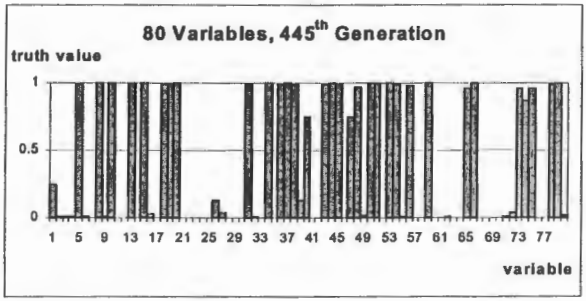
Figure 8. Fitness function (average in population and best individual) in successive generations; see description in text

5. Repairs of solutions: a recursive SAT version

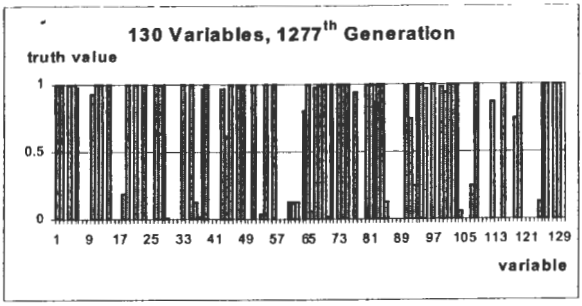
In this section, we discuss a problem the algorithm encounters at around the 190 variable mark. As has been shown, as the number of variables increased, so did the run time of the algorithm and the number of generations and the number of generations the algorithm had to iterate through to find the solution. Figure 9 shows the fuzzy values of the obtained solution for the SAT.



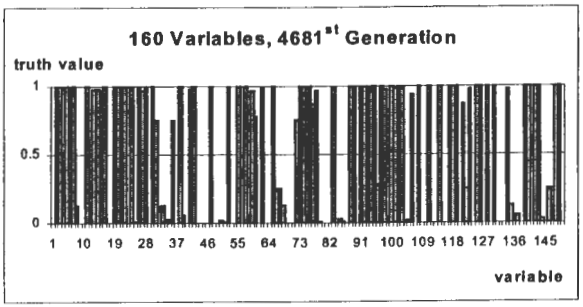
(a)



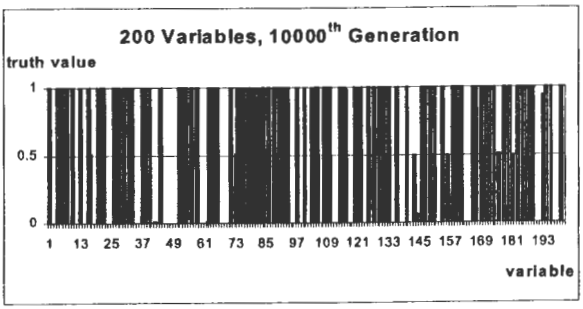
(b)


130 Variables, 1277th Generation
truth value
1
0.5
0
1 9 17 25 33 41 49 57 65 73 81 89 97 105 113 121 129
variable

(c)


160 Variables, 4681st Generation
truth value
1
0.5
0
1 10 19 28 37 46 55 64 73 82 91 100 109 118 127 136 145
variable

(d)


200 Variables, 10000th Generation
truth value
1
0.5
0
1 13 25 37 49 61 73 85 97 109 121 133 145 157 169 181 193
variable

(e)

Figure 9. Successive values for best solution at the end of the algorithm for (a) 50, (b) 80, (c) 130, (d) 160, and (e) 200 variables. All parameters, with the exception of population size and number of maximum generations, are as shown inTable 1.

As can be seen in Figure 9(e), the GA collapses at around 190 variables, and certain variables converge towards 0.5, while all the other values are essentially 0 or 1. (In some cases, the variable, though expressed in double precision format, had the value 1, 0, or 0.5.) A reaction to this was to introduce a more vigorous mutation, and less selective crossover parameter (that will allow for the most versatility). This, however, had no affect other than to accentuate the convergence of the variables toward one of the 3 values (0, 1, or 0.5). Another thought was that perhaps the random number generator was somewhat deficient[1]. However, a more vigorous random generating function did not solve the problem either. We therefore had to come up with a fixing mechanism.

5.1 First try: brute force method

Since at 200 variables, there were always 6-10 variables that did not converge, this is a small enough number to search using enumeration. A small problem was encountered in implementing this exhaustive search since it is not known at programming time how many variables would need to be fixed. This was solved using a binary representation: if there are m variables to fix, simply count to 2^m and the binary representation would indicate what value each variable should assume for this evaluation of the function.
The brute force method yielded the desired results in obtaining the final solution from one that looks like the one shown in Figure 9(e). After so many generations, the algorithm is stopped and each variable in the best individual is examined (it is assumed that the best individual at this point is closest to the actual desired solution). The cutoff points were arbitrarily decided to be at 0.1 and 0.9, at which point the variable is assumed to be 0 or 1 respectively. All other variables then go through the exhaustive search for the solution. Obviously, we might encounter a variable with a value of 0.93 that needs to assume a 0 for the solution, in which case the solution will never be achieved since it would get fixed at 1. However, experimentation showed that this does not happen, and this method works. Also, one can note that since any combination of values satisfying the Boolean function would be as good as the next (for a function with more than one minterm), there is no trap of a "local" optimum.

5.2 Second try: recursion

By the time the GA is trying to satisfy a 250 variable function, there are over 40 variables to fix, and so another method had to be used to resolve these variables. As was shown in Section 4, the brute force method is not very effective for solving more than 20-25 variables, and we were faced with far more than that, and expecting the number to grow

[1] The random function generator used was the C standard library rand() function. At first, the lower bits were used (with the modulus operator). Since all such functions are only pseudo-random, it is recommended to use the upper bits, which provides for longer cycles.

quite fast as the number of total variables in the function increased. It seemed that the number of variables that need to be resolved warranted their own GA, and so we looked to recursion.

The GA has now moved into a function instead of being the main body of the program, to which the Boolean function and a list of the indices of the variables that need fixing are passed. The GA then generates the appropriate population size and tries to satisfy the function. If it does not succeed, the unresolved variables are passed into the next level and the GA executes again. The end-point for the recursion is, of course, when the function has been satisfied, or there are fewer than 10 variables to resolve, at which point the problem is small enough for an enumerated search. A parameter "p" links the size of the population with the size of the problem, namely population size = maximum number of generations = p^* no of variables.

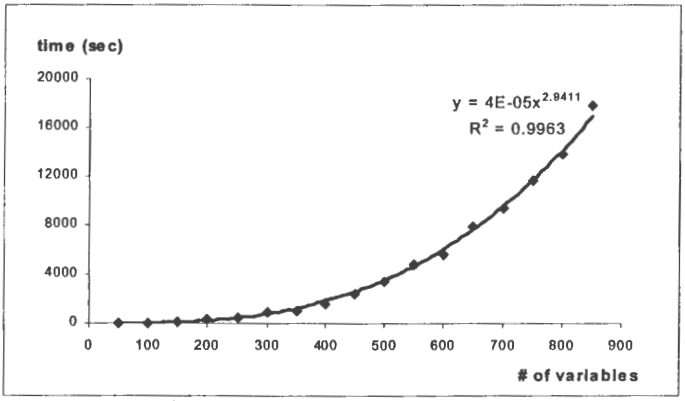


Figure 10. Graph of total run-time vs. number of variables in the function for the recursive GA

The recursive version of the algorithm proved to be both effective and efficient. The use of recursion allowed us to explore an even smaller search space than before and consequentially, the GA took less time. A comparison between Figure 10 and Figure 6 yields some rather interesting observations, the most obvious of which is the fact that the recursive GA function grows much slower than the regular one. It is also intriguing to note that the regular GA's run-time grows exponentially, while the recursive GA's run-time grows as a power of the number of variables. We have yet to find the upper limit of the recursive GA, but we do know (via experiments) it can handle the SAT for 1200 variable Boolean functions.

Total Number of Variables	Number of Unconverged Variables (in first level)	Population Size	Number of Generations	Total Time
800	269	3200	3200	3:50:25

500	127	2000	2000	0:57:07
300	70	1200	1200	0:15:58
100	7	400	400	0:00:37

Table 3. Performance of the recursive version of GA for different number of variables in the SAT problem

The levels of recursion the algorithm has to go through grows as the number of variables increase, as is to be expected. Table 3 illustrates the number of variables that have to be fixed for various sizes of the SAT. Considering the 800 variable SAT, one would treat the next level of the recursion as a 269 variables SAT, and solve the problem in this way. As can be seen from the table, an 800 variable function would require 2-3 levels of recursion to achieve the final solution. Figure 11. In general, the following observation holds: by increasing the size of the population, we tend to reduce the number of variables that need to be repaired at the second phase of the genetic optimization but increase the run time for each phase.

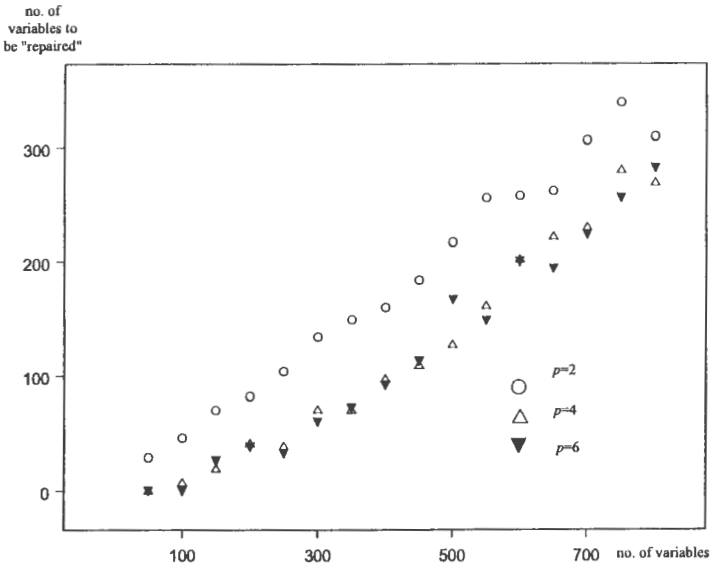


Figure11. Number of variables to be "repaired" at the second stage of the recursive GA as a function of total number of the Boolean variables in the problem (the parameter p links the size of the population with the size of the problem, namely $pop\_size = p^*$ no_variables)

6. Conclusions

The proposed embedding principle makes the original SAT problem continuous while fully retaining its logical nature. The original problem represented in the new search space was then solved using a standard version of the genetic algorithm. The study provides with yet another convincing example of a successful interaction between technologies of evolutionary computing and fuzzy sets underlining the importance of the main hybrid pursuit of computational intelligence [9]. GAs, especially the recursive version proved very efficient for handling multivariable SAT problems. It exhibits better run-time characteristics than its one-level counterpart. The thorough experiments revealed that the recursive GA can solve SAT problems with more than 1,000 variables.

Acknowledgments

The support from the Natural Sciences and Engineering Research Council of Canada (NSERC) and ASERC (Alberta Software Engineering Research Consortium) is gratefully acknowledged.

7. References

1. T. Bäck, U. Hammel, H.P. Schwefel, Evolutionary computation: comments on the history ad current state, *IEEE Trans. on Evolutionary Computation*, vol. 1:1, 1997, 3-17.
2. K. A. De Jong, W.M. Spears, Using genetic algorithms to solve NP-complete problems, In: *Proc. of the 3rd Int. Conf. on Genetic Algorithms*, Morgan Kaufmann Publ., San Mateo, 1989, pp. 124-132.
3. D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, Piscataway, NJ, 1995.
4. M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP - Completeness*, W.H. Freeman, San Francisco, 1979.
5. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, 1989.
6. J. Hartmanis, On computational complexity and the nature of computer science, *Comm. ACM*, 37, 1994, 37-43.
7. P. Mazumder, E.M. Rudnick, *Genetic Algorithms for VLSI Design, Layout & Test Automation,* Prentice Hall, Upper Saddle River, NJ, 1999.
8. Z. Michalewicz, *Genetic Algorithms + Structures = Evolution Programs*, 3rd edition, Springer Verlag, Heidelberg, 1996.
9. P. Pardalis, On the passage from local to global in optimization, In: J.R. Birge, K.G. Murthy (eds.), *Mathematical Programming*, The Univ. of Michigan Press, 1994.
10. W. Pedrycz, *Computational Intelligence: An Introduction*. CRC Press, Boca Raton, Fl, 1997.
11. W.G. Schneeweiss, *Boolean Functions with Engineering Applications and Computer Programs*, Springer Verlag, Berlin, 1989.
12. K. Wagner, G. Wechsung, *Computational Complexity*, D. Reidel, Dordrecht, 1986.