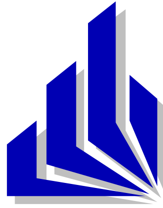


Universität Paderborn

Fakultät für
Elektrotechnik, Mathematik und Informatik

Studienarbeit

A Study of Evolutionary Algorithms for the Satisfiability Problem

Henning Ellerweg

vorgelegt bei

Prof. Dr. Hans Kleine Büning
Dr. habil. Benno Stein

Paderborn, Oktober 2004

Declaration

Hereby I certify that this work is the result of my own investigations and that to the best of my knowledge and belief it does not contain any material previously published except those listed in the text and the bibliography.

Deklaration

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig erarbeitet habe und sie nach besten Wissen und Gewissen keine bereits veröffentlichten Materialien enthält, außer die im Text und Literaturverzeichnis genannten.

H. Ellerweg
October, 2004

Acknowledgements

I would like to thank Chuan-Kang Ting for his support and founded criticism, which enabled me to write this work more comprehensible – by far more comprehensible.

Contents

1 Introduction **1**

2 Foundations **1**
- 2.1 The Boolean Satisfiability Problem . . . 1
- 2.2 Complete Algorithms . . . 2
- 2.3 Incomplete Algorithms . . . 3

3 Evolutionary Algorithms **5**
- 3.1 Basic Elements . . . 5
- 3.2 Types of Evolutionary Algorithms . . . 10
  - 3.2.1 Evolutionary Programming . . . 10
  - 3.2.2 Evolutionary Strategies . . . 11
  - 3.2.3 Genetic Algorithms . . . 11
  - 3.2.4 Genetic Programming . . . 11

4 Evolutionary Computation for SAT **12**
- 4.1 Representations . . . 13
  - 4.1.1 Bit-String Representation . . . 13
  - 4.1.2 MASKs . . . 15
  - 4.1.3 Clausal Representation . . . 17
  - 4.1.4 Path Representation . . . 18
  - 4.1.5 Floating Point Representation . . . 19
  - 4.1.6 Fuzzy Representation . . . 19
- 4.2 Selection . . . 20
- 4.3 Fitness Functions . . . 21
  - 4.3.1 SAW . . . 21
  - 4.3.2 Refining Functions . . . 23
  - 4.3.3 SAW and Refining Functions . . . 23
- 4.4 Genetic operators . . . 25
  - 4.4.1 Crossover . . . 25
  - 4.4.2 Mutation . . . 26
  - 4.4.3 Local Search . . . 27
- 4.5 Parallel-GA . . . 28
- 4.6 Performance Evaluation . . . 28

5 Conclusion **29**
- 5.1 Criticism on EA solving SAT . . . 29
- 5.2 Further Research Possibilities . . . 34

1 Introduction

The satisfiability problem (SAT) is a classic NP-complete problem and has been applied on practical problems like consistency check and circuit design. The approaches to solve SAT can be divided into two categories: Complete and incomplete methods. Complete methods include algorithms, based on the Davis-Putnam procedure which are able to decide if a given formula can be satisfied or not. Incomplete methods include evolutionary algorithms based on the simulation of natural mechanisms and Darwins "Survival of the Fittest". However they are not able to decide satisfiability because these algorithms typically cannot ascertain if a given formula is satisfiable at all.

This paper intends to give an overview of evolutionary computation for the satisfiability problem from its beginnings in 1989 (cf. [JoSp89]) until today. Moreover the general effectiveness of evolutionary computation for SAT will be discussed in the conclusion.

The paper is organized as follows: Section 2 gives a brief introduction to the SAT problem, the Davis–Putnam procedure and incomplete methods. Section 3.1 introduces a general structure of evolutionary algorithm and its operation is described. Section 3.2 presents four subfields of evolutionary computation. The focus of this paper is in section 4 presenting research ideas and experiments for evolutionary satisfiability solvers. Finally a summary of 15 years of research is given in the conclusion in section 5.

2 Foundations

This section gives a brief introduction to the satisfiability problem. Furthermore, the Davis-Putnam procedure is presented since it is a classical example of a complete method. As an instance of an incomplete method GSAT is introduced.

2.1 The Boolean Satisfiability Problem

The boolean satisfiability problem is formulated in the context of propositional logic formulas. Given a boolean formula F and a set of atoms $A = \{a_1, a_2, \ldots a_n\}$ with $a_i \in F$ for $1 \leq i \leq n$. The boolean satisfiability problem is stated as follows:

> Does there exist an atom truth assignment I, such that every $a_i \in A$ is whether set to $true$ or $false$ and I satisfies F i.e. $F(I) = true$.

Cook showed that this decision problem is an NP-complete problem [Co71]. A boolean formula F can be transformed into its logical equivalent *conjunctive normal form* (CNF). A formula F is in CNF if F is a conjunction of disjunctions and only atoms are allowed to be negated. Conjunctive normal form can be computed by iterated application of DeMorgan's Law and distributive law. However, the transformation to CNF may lead to an exponential increase of clauses. Consider a formula $F = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$. After transformation to CNF F', F' has 2^3 clauses:

$$(a_1 \vee a_2 \vee a_3) \wedge (a_1 \vee a_2 \vee b_3) \wedge (a_1 \vee b_2 \vee a_3) \wedge (a_1 \vee b_2 \vee b_3) \wedge$$
$$(b_1 \vee a_2 \vee a_3) \wedge (b_1 \vee a_2 \vee b_3) \wedge (b_1 \vee b_2 \vee a_3) \wedge (b_1 \vee b_2 \vee b_3)$$

k-CNF is a special case of CNF, where each clause contains at most k literals. It was also shown that any boolean formula can be converted to a formula in 3-CNF for which the satisfiability problem is still NP-complete. In general it is hardly possible to transform a given boolean expression into 2-CNF, whereas 2-CNF is in P. The MAXSAT problem is derived from SAT and CNF. It is an optimization problem and asks for an atom truth assignment I such that a maximum number of clauses is satisfied. A more detailed description about propositional logic and the satisfiability problem can be found in in [BüLe94].

2.2 Complete Algorithms

A *complete* algorithm for SAT is an algorithm which can determine if a formula is satisfiable or not. Many complete algorithms solving SAT are based on the Davis-Putnam procedure [DaPu62]. Its pseudocode is presented in figure 1.

Davis-Putnam(CNF-Formula $\boldsymbol{\alpha}$)
  if ($\boldsymbol{\alpha}$ has no more clauses) *return true*
  if ($\boldsymbol{\alpha}$ has an empty clause) *return false*
  if ($\exists$ unit-clause $\boldsymbol{c_i} \in \boldsymbol{\alpha}$ with literal $\boldsymbol{l_i}$)
    return **Davis-Putnam**($\boldsymbol{\alpha_{[l_i/1]}}$)
  $\boldsymbol{l_s} \longleftarrow$ choose a literal by some strategy s
  if (**Davis-Putnam**($\boldsymbol{\alpha_{[l_s/1]}}$) = *true*)
    return true
  return **Davis-Putnam**($\boldsymbol{\alpha_{[l_s/0]}}$)

Figure 1: The Davis-Putnam Algorithm

This algorithm recursively searches for a satisfying truth assignment of the atoms a_i of α. To this end it sets a literal l_i to a truth value, for example l_i is set to *true*. The chosen truth value alters the given formula. A clause c_s which contains a positive occurrence of l_i is deleted because of CNF and since it is satisfied by l_i. All negative occurrences of l_i in clauses c_u are deleted. This leaves fewer literals in such a clause c_u, so that less literals are able to satisfy c_u. Clauses are not altered by the truth assignment, if they do not contain the set atom. This transformation is denoted by $\alpha_{[l_i/t]}$, t a truth value, which results in a formula with the properties described above.

Since α is in CNF, a satisfiable truth assignment is found if a reduced α does not have any clause left, because then every clause of the initial formula is satisfied by the current truth assignment. The recursion will not search further if the formula contains a clause with no literals (an empty clause) because this clause cannot be satisfied in such a search branch. This behavior is guaranteed with lines (1)-(2).

A *unit-clause* is a clause containing only one literal. This literal has to be set to the truth value which satisfies the clause. Otherwise an empty clause would be generated and the search branch would be aborted in the next recursive step. Thus Davis-Putnam may find a satisfiable truth assignment of α in a search branch with satisfied unit-clauses. If no such clause exists, another selection criterion – strategy s – must be provided to decide which literal is set next. A lot of researches focus on the design of s because a well-designed strategy can greatly improve the search in terms of needed recursions and time. Nevertheless, if strategy s picks a literal and the resulting formula is not satisfiable, the formula generated with the same literal but set to false has to be searched. This ensures that this procedure is a complete algorithm.

2.3 Incomplete Algorithms

Incomplete algorithms for SAT are able to decide if a given formula is satisfiable by finding a satisfiable truth assignment. However, they cannot ascertain unsatisfiability, since they have no means to detect contradictory formulas. Nevertheless, the loss of completeness of incomplete methods is often compensated by a speedup in comparison to complete methods, because the procedure does not need to search a tree and is allowed to jump in the search space.

One famous incomplete algorithms for SAT is GSAT, presented by Selman, Levesque and Mitchell [SLM92]. Its pseudocode is given in figure 2. GSAT starts its search with a randomly generated truth assignment for α. There-

GSAT(CNF-Formula $\boldsymbol{\alpha}$, **MAX-FLIPS**, **MAX-TRIES**)
  for **i** ← **1** *to* **MAX-TRIES**
    T ← a randomly generated truth assignment for $\boldsymbol{\alpha}$
    for **j** ← **1** *to* **MAX-FLIPS**
      if **T** satisfies $\boldsymbol{\alpha}$ *then return* **T**
      v ← an atom such that a change in its truth
        assignment of **T** gives the largest increase of
        satisfied clauses in $\boldsymbol{\alpha}$, ties of largest increase
        are randomly broken
      T ← **T** with the inverse truth value of **v**
  return 'no satisfying assignment found'

Figure 2: The GSAT Algorithm

after the greedy strategy attempts to maximizes satisfied clauses, which leads the search to more promising regions of the search space by satisfying more and more clauses. The increase may be equal to zero, so that the search is able to perform *sideway moves* (cf. [SLM92]). If a satisfiable truth assignment is found the search is halted and T is returned. Since this process is not guaranteed to find a solution, it is terminated after MAX-FLIPS *truth-assignment-flips*. Then the search starts again with a randomly generated T. This process repeats until MAX-TRIES is exceeded and no solution was found.

GSAT's greedy strategy to flip a v with the largest increase of satisfied clauses, has a major disadvantage. A formula can be designed in such a way that the greedy mechanism is guided away from the solution. Consider the formula given in figure 3.

$$
\begin{array}{llll}
(a_1 \vee \neg a_2 \vee a_3) & \wedge & (a_1 \vee \neg a_3 \vee a_4) & \wedge \\
(a_1 \vee \neg a_4 \vee \neg a_2) & \wedge & (a_1 \vee a_5 \vee a_2) & \wedge \\
(a_1 \vee \neg a_5 \vee a_2) & \wedge & (\neg a_1 \vee \neg a_6 \vee a_7) & \wedge \\
(\neg a_1 \vee \neg a_7 \vee a_8) & \wedge & (\neg a_1 \vee \neg a_9 \vee a_{10}) & \wedge \\
(\neg a_1 \vee \neg a_{10} \vee a_{11}) & \wedge & (\neg a_1 \vee \neg a_{11} \vee a_{12}) & \wedge \\
(\neg a_1 \vee \neg a_{12} \vee a_{13}) & \wedge & \ldots & \wedge \\
(\neg a_1 \vee \neg a_{98} \vee a_{99}) & \wedge & (\neg a_1 \vee \neg a_{99} \vee a_6) &
\end{array}
$$

Figure 3: A SAT formula which is difficult for GSAT

This formula is satisfied only if a_1 is set to true[1]. The greedy strategy tends to set a_1 to false because this setting satisfies many clauses. Hence, a_1 set to false has a large increase of satisfied clauses and is therefore unable to find a solution.

Selman and Kautz overcame these problems by their improvements to GSAT [SeKa93]. They introduced *clause weights* for clauses. The weights are adapted during search so that the greedy strategy favors other clauses after some time. Clause weights are initialized to 1 and after each try the weight of a clause is increased by k if it is unsatisfied after MAX-FLIPS flips. This has a desired effect that clauses remaining unsatisfiable for many tries can collect more weights. In their experiments most often $k = 1$. The greedy strategy then was adapted so that a clause c_i with some weight w_i has an increased gain of w_i, instead of 1, if satisfied. Referring to the example described above the first 5 clauses will get so much weight during search (provided that MAX-TRIES is big enough) that the greedy strategy will pick atom a_1 because it has the highest increase.

Selman and Kautz also introduced the *average-in* and *random walk* strategy in [SeKa93] as well, both further improve the original GSAT. Another improvement to clause weights was proposed by Frank in [Fr96] where clause weights are adapted after each truth-assignment-flip. Furthermore, Frank implemented an adaptive strategy to increase k for the following reason. A clause weight increase of 1 at the beginning of the search has high influence because it doubles the clause weight. However, after some time an increase of 1 is rather ineffective since clauses may already have high weights. A brief list of other GSAT variants and improvements is given in [ScSo00] by Schuurmans and Southey.

3 Evolutionary Algorithms

3.1 Basic Elements

The main concept of evolutionary algorithms is a computational model based on Darwin's theory: *Survival of the fittest*. For better understanding of the functioning of evolutionary algorithms the pseudocode of a general EA is given in figure 4 and its elements are described in the following.

[1]If a_1 is set to false, then a_2 must be set to true because of clauses c_4 and c_5. If a_2 has to be true then a_3 and a_4 has to be true because of clauses c_1 and c_2. Since a_4 has to be true a_2 must be set to false (c_3), which is a contradiction

Evolutionary Algorithm(Problem $\boldsymbol{\pi}$)
Initialize a population $\boldsymbol{\phi}$ according to $\boldsymbol{\pi}$
Calculate fitness of each individual $\in \boldsymbol{\phi}$ according to $\boldsymbol{\pi}$
while (Terminating condition is not met) *do*
Parents ← **Selection**(ϕ)
Offspring ← **Genetic Operators**(**Parents**)
Calculate fitness of each individual ∈ **Offspring** according to $\boldsymbol{\pi}$
ϕ ← **Survival**(ϕ, **Parents**, **Offspring**)

Figure 4: General Evolutionary Algorithm

The task of an EA is to solve a given problem π, which is most often, but not limited to, an optimization problem. A typical example of an EA problem is the MAXSAT problem, but it can even be as complex as the automatically evolution of program code (cf. section 3.2.4). One can see from the given pseudocode that population initialization and fitness calculation are both problem dependent. However, EA search is performed by selection, genetic operators and survival. All three are problem independent. Nevertheless, the performance of selection and genetic operators can be improved by exploiting domain knowledge (cf. 4.2, 4.4).

An evolutionary algorithm has a *population*, which is composed of one or more *individuals*. An individual, also called *chromosome*, consists of *genes*, which are the variables for an EA to alter during searching for a solution of the given problem. A gene can be a single bit, a decimal, a floating point number, a tree, or a part of program code. The important thing about a chromosome is that it can be an *encoding* of a solution to the given problem. An encoding maps a potential solution from the problem domain, called *phenotype space*, to a chromosome in the search space of an EA, called *genotype space*. Every EA design has to undergo this *representation* issue. Consider the following example:

Encoding table	Vehicle	Color	HP	Cylinders
00	Car	red	50	4
01	Motorbike	blue	100	6
10	Pick-up	green	150	8
11	Truck	orange	200	12

A chromosome of this problem space is a concatenation of the 2-bit tuples and could look like this:

$$10011101$$

The above bit-string represents a pick-up, because its first 2 bits are 10. Furthermore, the pick-up has blue color (01), 200 HP (11), and 6 cylinders (01). The population is usually randomly initialized, although there is no reason why one should not use biased or pre-searched individuals.

The *fitness function* is the essential of the algorithm. It can be considered as a mapping function from the genotype to the phenotype space and measures the solution-quality of a chromosome. Assume that the solution of the vehicle example is a orange truck with 8 cylinders and 200 HP. Then a fitness function could return the number of matches encoded in a chromosome. According to this fitness function, a (Pick-up, orange, 200, 8)-chromosome would have a fitness of 3, while (Car, red, 50, 4) would score 0. The fitness function acts as a means to differentiate individuals according to the extent of their contribution to a solution. Therefore, it should not return binary values. It is not always possible to derive a fitness function which leads to the global optimum, but to a local one. However, its implementation can range from a simple mathematical function to the execution of a simulation. The quality of the fitness function determines how effective the search of an EA will be.

The *selection* of parents is used to obtain a computational model of survival of the fittest. The fitter the individual in the population, the higher the likelihood that it is selected to act as a parent. A common implementation of survival of the fittest is the *roulette wheel selection*. This selection scheme sets up a roulette wheel, such that for each individual an interval is reserved on the wheel. The size of the interval is:

$$\frac{fitness(individual)}{\sum\limits_{population} fitness(individual)},$$

so that individuals of higher fitness get more space. These intervals are arrayed one to the other on the wheel. By picking a random number r, $0 \leq r \leq 1$ exactly one of the intervals, i.e. an individual, is chosen. The usage of a random number corresponds to the spinning of the roulette wheel. Due to the design of the roulette wheel survival of the fittest is achieved, because fitter chromosomes have more space (chance) on the wheel. Another implementation is called *tournament selection*, which holds tournaments of n individuals. The fittest wins the tournament and therefore becomes a parent. Note that roulette wheel and tournament selection are both *fitness proportional selection* schemes, since fitter individuals are favored. Other selection

schemes are presented in Whitley's overview [Wi01]. Often those selection mechanisms are repeated until a desired number of parents are selected.

Once a set of parents is selected, the parents are *mated* to reproduce offspring. Mating is done by the application of genetic operators to the parents. Since the operators are working on the chromosomes of a population and the chromosomes possess a representation, the operators have to be designed appropriate to the representation. The two most widely used operators are *crossover* and *mutation*.

The idea of crossover is to take two or more parents (most often different to each other to avoid clones) and recombine their genes to get new offspring. The amount of offspring depends on the crossover scheme. A simple example of so called one-point-crossover is depicted in the following:

Parent 1:	00110101
Parent 2:	10101010
Crossover point:	between 4th and 5th bit
Offspring 1:	00111010
Offspring 2:	10100101

A crossover point is chosen randomly between two bits, thereby guaranteeing that no clones are produced. Then the chromosomes of both parents are cut at that point, such that 4 bit-string pieces are generated. The chromosome Offspring 1 gets the left part of Parent 1 and the right part of Parent 2, whereas Offspring 2 is set to the left part of Parent 2 and the right part of Parent 1. In a nutshell, crossover rearranges pieces of chromosomes.

This scheme can be easily extended to the also popular two-point-crossover (choose randomly two crossover points) or uniform-crossover (for each gene choose randomly the parental source of inheritance). Since both parents are encoding a part of the solution to a given problem, the recombination in form of the offspring may then hold the solution.

The mutation operator is applied to a chromosome and changes one or more randomly chosen genes. In terms of the crossover example, mutation would flip a bit from 0 to 1 or vice versa. The result of such a flip can be a solution to the problem.

Both described operators are able to generate an individual solving the given problem. Therefore it suffices to use either crossover or mutation as sole genetic operator. However, the use of crossover without mutation arises a

problem. Consider a population of bit-strings as described in the crossover example and suppose that for every chromosome in the population the last bit has a value of 0. Then the crossover operator is not able to generate an offspring which has a value of 1 in its last gene. If this bit is crucial for the problem solving, an EA utilizing only crossover will never find a solution. This phenomenon is known as *genetic drift* [RoPr99]. Therefore crossover is most often used in combination with mutation to achieve a diverse population, that is a population with many different chromosomes, and to recover *lost* genes.

The utilization of genetic operators is usually controlled by stochastic means. If crossover seems to be the driving force during the search, crossover should get a high probability of usage and the same is true for mutation. The application rate also needs to balance crossover's exploitation of partial solutions versus mutation's exploration by flipping randomly chosen genes.

The described methods of crossover and mutation are *blind*, because they are manipulating the chromosomes without considering the problem space. But there are *informed* operators as well, which utilize problem domain knowledge to improve the offspring reproduction process. Section 4 describes some informed operators for the satisfiability problem space.

There is a lot of advanced literature which tries to improve the performance of EAs by supplying other means of recombination. For example the gene pool recombination by Mühlenbein and Voigt [MüVo95] or the utilization of self-adapting crossover schemes by Spears [Sp95].

After the offspring is produced, the population for the next iteration is calculated by a *Survival* function. This *next generation* can be composed by applying different strategies to the whole population, the parents, and the offspring. The offspring may displace the whole former population, but the usage of elitism is also allowed. Elitism prevents the loss of discovered individuals with high fitness, so that inferior offspring is rejected for the new population. After survival the whole process of selection, recombination, and survival is carried out again, until some terminate condition is met.

A common problem in EAs is *premature convergence* caused by a population consisting of many similar individuals. In an overearly stage of searching the population is not *diverse* so that crossover is performed on very similar chromosomes. Therefore mutation becomes the search's driving force. In other words, the population has *converged* to almost the same chromosomes, but this convergence is *premature* because none of the chromosomes is near the optimum. To counter this problem many algorithms maintain a diverse population, e.g. by using a high mutation rate.

One can see now how an EA is searching for a solution, namely searching the genotype instead of the phenotype space by applying genetic operators on fit individuals. Searching the genotype space is an advantage of an EA, because it is able to search in another representation space of the problem. However, this advantage comes at some cost. Since the search in the genotype domain is not a complete one – because the genetic operators do not provide means to accomplish that – the whole EA is an incomplete search method. Therefore an artificial terminate condition has to be supplied to an evolutionary algorithm. Such a condition is most often relying on a maximum number of fitness evaluations or on a maximum number of constructed generations. Thus the algorithm can be aborted with no solution found. For problems where a solution can be identified by the fitness function, the search can be terminated exactly when a solution is discovered.

The computational costs of an EA largely depends on the design of the fitness function. More detailed introductions can be found in the papers by Spears, De Jong, Bäck, Fogel and de Garis [SJBFG93] and Schoenauer and Michalewicz [ScMi97].

3.2 Types of Evolutionary Algorithms

Presently the research field of evolutionary algorithms is commonly divided into four research sectors: Evolutionary programming (EP), Evolutionary Strategies (ES), Genetic algorithms (GA), and Genetic Programming (GP). All four fields are briefly introduced in the following.

3.2.1 Evolutionary Programming

In 1966 Fogel et. al. (cf. [SJBFG93]) developed EP to evolve finite state machines. Typically, individuals are designed fitting to the problem without using a genotype space. Thus EP chromosomes are a model of the phenotype space, e.g. an EP algorithm for real-valued problem optimization would use real-valued genes composed in a real-valued vector, whereas traveling-salesman problems could be represented with ordered lists. EP does not use any selection scheme to separate fitter from less fit individuals because the whole population is used to produce the next generation. Only mutation (which is problem specific due to the problem specific representation) is applied to each of the parents. If population size is N, parents and offspring together have size $2N$. The $2N$ chromosomes form the successor generation by using a probabilistic elitism strategy. Thus it is more likely that individuals with high fitness are inserted in the next generation than individuals with bad fitness (cf. [SJBFG93]).

3.2.2 Evolutionary Strategies

1973 Rechenberg introduced ES utilizing a population composed of only one real-valued chromosome and therefore mutation is used as sole genetic operator. Schwefel extended the population to multiple individuals. Nowadays two different types of ES exist, namely:

- $(\mu + \lambda)$-ES
- (μ, λ)-ES

The terms in brackets stand for μ parents reproducing λ offspring. The difference between both is denoted by delimiter '+' or ',' representing the survival scheme. The symbol '+' represents the elitism strategy on both generations (elitism applied to the *sum* of parents and offspring) whereas ',' denotes that the best individuals in offspring will form the next generation, and because of that, $\lambda \geq \mu$ has to be valid. The real-valued gene mutation is often implemented by using a Gaussian mutation utilizing standard deviation σ (cf. [ScMi97]).

3.2.3 Genetic Algorithms

Holland introduced GA with bit-string encoding in 1975. Typically GA use bit-string encodings, but in recent publications also problem-specific representations are used (see below). Most often GA rely on fitness proportional selection utilizing roulette wheel or tournament selection. Mutation and crossover are usually performed but crossover is considered as the driving force of evolutionary search (cf. [ScMi97]). The original GA is not provided with an elite strategy so that a successor generation could have a deterioration in fitness.

3.2.4 Genetic Programming

GP was introduced by different authors solving different problems but their work was reconciled with the generic term GP (cf. [SJBFG93]). The task of GP is to evolve program code. Koza, for an instance, used a set of LISP S-expressions as genes. Usually one of these expressions is a compound-statement and/or an *if-then-else*-statement such that S-expressions can be combined in a chromosome of varying size. The initial population is composed of random programs. Parents are chosen by tournament selection which are then recombined with a crossover operator. Note that the swapping of S-expressions generates a valid S-expression, i.e. a valid program (cf. [ScMi97]). The fitness of such a program is evaluated by simulation, because

it has to be verified how *good* the evolved program can solve the problem. In general, GP is a more complex subtype compared to the other three evolutionary algorithms.

A GP example: An ant is able to go forward, turn left, turn right, detect adjacent food sources, pick up food, and return home. In addition this GP uses a compound, an if statement, and logic operators. The first generation may have the two programs as sketched in figure 5.

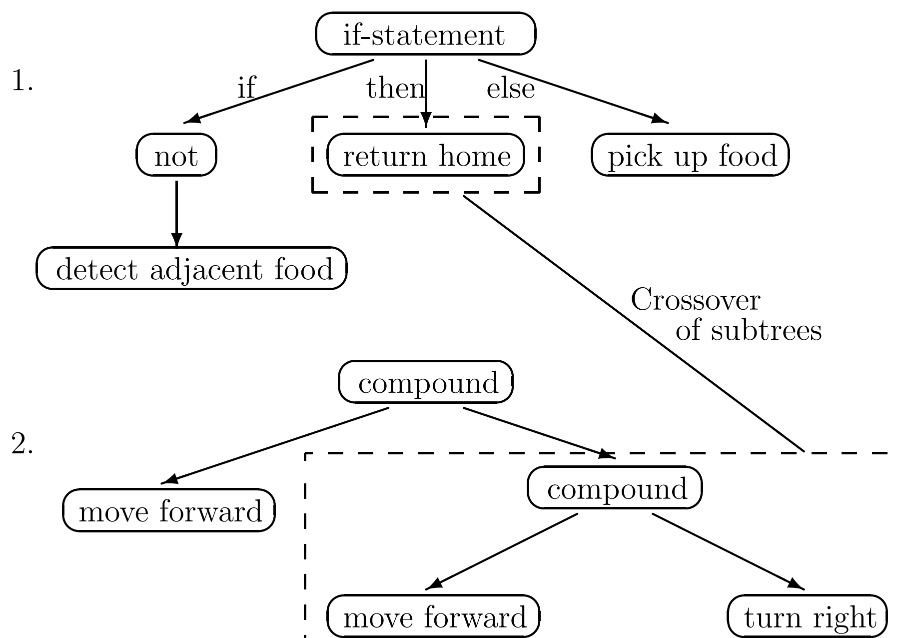


Figure 5: Genetic Programming Example

Performing a crossover could result in the program as depicted in 6.

After several generations the ant may be programmed in such a way that it looks for food and carries it back home. It can be very time-consuming to simulate these evolved programs to get a fitness value. Nevertheless, fitness values are needed since survival and fitness proportional selection are employed.

4 Evolutionary Computation for SAT

The intention of this section is to give an overview of EA research solving satisfiability problems. Because of the amount of publications this section is divided in some parts. At first, different kinds of representations and

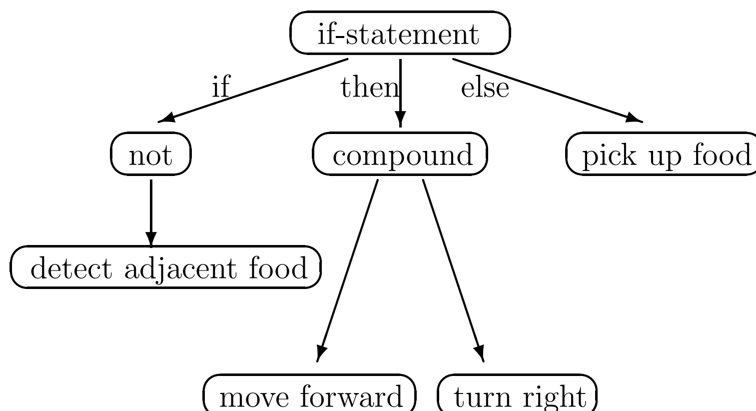


Figure 6: Genetic Programming Example

their representation dependent fitness functions and genetic operators are introduced. Thereafter a brief discussion of another approach to selection for the bit-string representation (see below) is given, followed by an examination of fitness functions introduced in the context of SAT. Finally improvements to genetic operators are reviewed and a parallel implementation of GA is sketched.

4.1 Representations

4.1.1 Bit-String Representation

De Jong and Spears' work [JoSp89] is one of the first published genetic algorithm explicitly designed to solve SAT. In their article they concentrated on a GA processing boolean formulas, not necessarily in CNF. In addition, they gave remarks on how to design a fitness function for CNF formulas which became popular in the aftermath. Therefore both fitness functions are presented here.

For representation they chose a bit-string of length N, where N is the number of atoms of the formula. Each single bit refers to the truth assignment of exactly one atom. If a bit is set to 0 (resp. 1) the atom is evaluated to $false$ (resp. $true$). Such a bit-string corresponds to an interpretation of a boolean formula (cf. [BüLe94]) and consequently it can be verified if this bit-string satisfies the formula or not.

De Jong and Spears claimed in their article:

> "It is hard to imagine a representation much better suited for use

> with GAs: it is fixed length, binary, and context independent in the sense that the meaning of one bit is unaffected by changing the value of other bits ...” ([JoSp89])

However this representation does not capture any intrinsic relations between variables and clauses as e.g. clausal representation [Ha95] or the path representation [GoVo97].

For choosing a fitness function appropriate to the established representation De Jong and Spears proposed several possibilities. Using the boolean formula itself can be discarded immediately, because its results are binary. As aforementioned, such a binary fitness function is not advantageous for EAs (cf. section 3.1 page 7). Alternatively they rely on the following fitness function f, which is computed after calculating the formula's truth values induced by the current bit-string.

$$\begin{aligned} f(true) &= 1 \\ f(false) &= 0 \\ f(\neg\alpha) &= 1 - f(\alpha) \\ f(\alpha \vee \ldots \vee \omega) &= max\,\{f(\alpha), \ldots, f(\omega)\} \\ f(\alpha \wedge \ldots \wedge \omega) &= avg^x\,\{f(\alpha), \ldots, f(\omega)\}, \end{aligned}$$

with avg^x denoting the average function to the power of x.

Nevertheless they encountered several difficulties with the presented function f, because logically equivalent formulas may have different fitness values due to the computation of the average function[2]. For this reason they tried to come up with a fitness function which complies with such a property, but they had no success. Due to this failure De Jong and Spears aimed at a fitness function holding true for truth invariance that is:

$$f(chromosome) = 1 \Rightarrow formula(chromosome) = true$$

This property could not be established either for general boolean formulas. However, this problem was overcome by applying De Morgan's law to the given formula in a preprocessing step such that only variables are allowed to be negated (cf. [JoSp89]).

The advantage of avg^x with $x > 1$ is that $\wedge$-clauses are getting a higher fitness value if they are more nearly *true*. But a too big x would result in

[2] $f(a_1 \wedge (a_2 \wedge a_3)) \neq f((a_1 \wedge a_2) \wedge a_3)$ e.g. $x = 1$, a_1 set to 0, a_2 and a_3 set to 1 then $f(a_1 \wedge (a_2 \wedge a_3)) = 0.5 \neq 0.75 = f((a_1 \wedge a_2) \wedge a_3)$

a behavior similar to a *min* function, which would only give binary results again. The authors empirically discovered a threshold of $x = 2$, since a higher value of x results in more needed fitness evaluations.

Considering boolean formulas in CNF, De Jong and Spears proposed to use a fitness function corresponding to the MAXSAT problem. Fitness calculation according to the MAXSAT problem is defined as follows:

$$f_{MAXSAT}(chromosome) = \sum_{clauses\ c_i} T(c_i, chromosome)$$

with

$$T(c_i, chromosome) = \begin{cases} 1 & \text{if } c_i \text{ is satisfied by chromosome} \\ 0 & \text{otherwise} \end{cases}$$

f_{MAXSAT} is used very often for evolutionary algorithms solving satisfiability, because it is easy to implement. In addition, it establishes truth invariance since if all clauses are satisfied the formula is satisfied as well. Note that f_{MAXSAT} yields the same fitness value for individuals which satisfy a different set of clauses. Thus an EA is not able to discern between chromosomes with the same fitness. However, it is of importance to distinguish between such chromosomes because not all of them may lead to better areas of the search space (cf. [GoVo97]).

Both fitness functions may run into difficulties solving SAT efficiently. Consider the formula depicted in figure 3 page 4. Recall that this formula can only be satisfied if a_1 is set to 1. Chromosomes with bit a_1 set to 0 have a high fitness value because a high percentage of the formula is satisfied. Therefore it is more likely that fitness proportional selection chooses two chromosomes both having bit a_1 set to 0. For such chromosomes, it is not possible to find a solution by applying solely crossover. If mutation alters bit a_1 after such a crossover is performed, a solution may be found though. However, note that the clauses containing $\neg a_1$ are of a design such that they form a logical implication chain. Crossover of chromosomes with bit a_1 set to 0 cannot evolve the implication chain of the given formula, because these fitness functions do not indicate the need to evolve this chain since the clauses are already satisfied.

4.1.2 MASKs

Hao and Dorne [HaDo94a] introduced the MASK method, which uses a bit-string representation as well, but it is extended by a *non-fixed* value '*'.

During evolution only these non-fixed values are altered and become fixed. A MASK-chromosome of length N, N is the number of atoms, is defined as:

$$F_1 \dots F_k \; W_1 \dots W_{N-k},$$

where F_i is a fixed value of 0 or 1 and W_j is a *wildcard* '*'. The population is initialized with a set of 2^m masks, $m > 0$, and the first m positions of each mask are initialized to the binary numbers between 0 and $2^m - 1$. For example if $N = 8, m = 2$ the initial population looks like this:

$$\{00*^6, 01*^6, 10*^6, 11*^6\}$$

Fitness evaluation is based on the fitness function by De Jong and Spears, but is extended to handle the wildcards. To this end Hao and Dorne use a set RGS[3] of randomly initialized bit-strings without wildcards. For their experiments then, they use a fitness function defined as

$$f_{mask}(m) = \underset{x \in RGS}{avg} \{f(m_{[i/x_i]})\},$$

where f is the fitness function introduced by De Jong and Spears and $m_{[i/x_i]}$ is the resulting bit-string if every wildcard of m at any position i is replaced by the corresponding ith value of bit-string x ∈ RGS. Depending on the size of RGS the fitness of a mask is the average of many fitness evaluations performed by f. Note that by using this fitness function a boolean expression underlies the same restrictions as they are described in section 4.1.1.

Selection takes the better (fitter) half of the evaluated population, the rest is discarded. As a genetic operator a *divide* operator is introduced which generates two offspring per mask by setting the first wildcard W_1 to 0 resp. 1, therefore reestablishing a population of size 2^m. This process is repeated until all wildcards are fixed. If such a fully instantiated mask has a fitness of 1, it is a solution due to the use of f. In fact Hao and Dorne stop evolving the population if a mask's wildcards matched with a bit-string from RGS is a solution.

Hao and Dorne gave a detailed comparison between De Jong and Spears' GA and their MASK Method in [HaDo94b]. The MASK method clearly outperformed the GA in terms of needed number of fitness evaluations. As reasons for this behavior Hao and Dorne claim that the recombination of different high fitness chromosomes is not effective for finding a solution. This is because the assumption that recombination of 'good' solution parts (encoded

[3] Randomly generated bit-strings

in the representation) forms a 'better' solution is wrong in terms of SAT (cf. [HaDo94b]). Another reason is that a mask is covering a subtree in the search space, whereas a bit-string is exactly one branch of it. Crossover and mutation produce jumps of such a branch, while the dividing procedure in combination with a fitness function utilizing RGS is more search alike.

4.1.3 Clausal Representation

Hao's clausal representation [Ha95] is an alternative to standard bit-string representation. This encoding tries to use inherent information of CNF expressions and is therefore restricted to them. The clausal representation is not composed by one gene per atom, but by smaller bit-strings for each clause. A clause's bit-string has k bits if the clause is of size k. Moreover, such a bit-string is only allowed to be instantiated to values such that the clause is satisfied by these values. Consider the following formula:

$$(a_1 \vee \neg a_2 \vee a_3) \wedge (\neg a_1 \vee \neg a_2 \vee a_3)$$

Then the clauses' bit-strings are permitted to have the following values:

$$\begin{aligned}(a_1 \vee \neg a_2 \vee a_3) &\Rightarrow \{000, 001, 011, 100, 101, 110, 111\}\\ (\neg a_1 \vee \neg a_2 \vee a_3) &\Rightarrow \{000, 001, 010, 011, 100, 101, 111\}\end{aligned}$$

A k-clause has $2^k - 1$ valid instantiations. Such a bit-string is locally consistent with its clause, but it may cause global inconsistence. Consider the example formula from above and suppose the formulas bit-string is set to

$$001 \oplus 101,$$

where $\oplus$ denotes the concatenation of both bit-strings. This string causes a variable inconsistency, because 001 requires a_1 to be set to 0, but 101 necessitates a_1 to be set to 1. If there is no inconsistency a satisfying solution is discovered, because of the restricted instantiation of clause bit-strings.

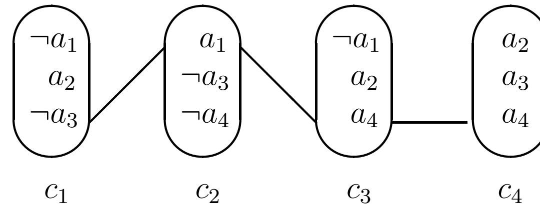
Hao suggested four different fitness functions all following the same idea: Minimizing the number of inconsistent variables. Crossover and mutation are designed in such a way that they do not create local / clause inconsistencies. He also introduced a simple local search procedure to direct the search to promising areas. For the starting population Hao recommended to use $2^k - 1$ individuals for k-CNF, since then all possible clause instantiations could be distributed among the chromosomes.

In the end he provided neither any performance evaluation for his clausal

representation nor is there any successor work using clausal representation. A problem arises from the size of clausal representation, since it usually needs much more bits compared to standard bit-string representation. Thereby it increases the genotype search space. For example, hard random boolean formulas have more clauses than atoms because of the *transition phase* at 4.3 clause to variable ratio (cf. [SML96]). Thus a chromosome of clausal representation for a hard random k-SAT formula has $4.3 * k$ more genes than a bit-string representation chromosome.

4.1.4 Path Representation

Gottlieb and Voss [GoVo97] proposed a path representation, which is similar to Hao's clausal representation. Path representation assumes a boolean expression in CNF. The basic idea is to choose one atom from each clause and instantiate it, such that the clause is satisfied. Since variables can occur in different clauses, variable inconsistencies may occur. Note that no inconsistency signifies a solution. A path chromosome may have the following design:



Thus a chromosome of this path would be represented by:

$$(\neg a_3, a_1, a_4, a_4),$$

which is valid a solution. Another chromosome for this example formula could be $(\neg a_3, a_1, a_4, a_3)$, which has a variable inconsistency i.e. a_3. Therefore the authors established a fitness function favoring less inconsistencies in a chromosome. Crossover for path representation is similar to uniform-crossover. It takes two chromosomes as input and processes each gene but exchanges genes only under some preset probability[4]. Then, the current gene is checked if gene a of parent A inflicts fewer inconsistencies in A than the corresponding gene b of parent B in B. If so, b is set to a otherwise a is set to b. Mutation, on the other hand, processes every clause by constructing a random set of possible values for that clause. The mutation procedure picks the value that causes the fewest inconsistencies.

[4]This probability check is not applied in uniform-crossover

Gottlieb and Voss experimented with this design and compared it to the standard bit-string version. The results indicate that their design needs more fitness evaluations until a satisfying assignment is found. Moreover it has a worse success rate finding a solution.

4.1.5 Floating Point Representation

Leaving the realm of discrete bits, Bäck, Eiben and Vink [BEV98] proposed a floating point representation. The conception of floating point representation is to replace each literal x by a floating point variable x_f in the interval $[-1, 1]$, with -1 meaning $false$ and 1 resembling $true$. The design of the fitness function follows the structure of the given formula. A positive literal x is replaced by $(x_f - 1)^2$ and negative ones by $(x_f + 1)^2$. The boolean operators $\vee$ and $\wedge$ are simulated by $*$ resp. $+$. Consider the following example:

$$(a \vee \neg b \vee c) \wedge (b \vee c \vee \neg d) \wedge (a \vee \neg d),$$

which results to a floating point fitness function:

$$f_{float} = (a_f - 1)^2(b_f + 1)^2(c_f - 1)^2 + (b_f - 1)^2(c_f - 1)^2(d_f + 1)^2 + (a_f - 1)^2(d_f + 1)^2$$

A solution is found if all variables x_f become an appropriate solution value of 1 (resp. -1) and the assignment results in $f_{float} = 0$. The fitness function f_{float} can be viewed as a continuous optimization problem.

The population is initialized with random values $\in [-1, 1]$ and the authors rely on rounding to -1 resp. 1 during the evolutionary process to check if a solution was found. Since the authors implemented this approach for evolutionary strategies (cf. section 3.2.2), they used mutation as sole genetic operator. Mutation uses standard deviation σ, which is bounded by a maximum value of 3.0 and Bäck et. al. applied three types of recombination called *discrete recombination*, *global intermediate recombination* and *intermediate recombination*. Later experiments have shown that the floating point representation using ES performs worse than a GA, which is referred to SAWEA [BEV98].

4.1.6 Fuzzy Representation

Pedrycz, Succi and Shai [PSS02] extended the floating point representation to a fuzzy representation. The authors follow Bäck et. al. ([BEV98]) intent to transform the discrete search space of boolean formulas to a continuous search space by utilizing fuzzy sets and fuzzy logic. An advantage of fuzzy logic is that this representation preserves the logic characteristics of boolean formulas.

With this idea they propose a whole class of fitness functions, since they claim that any fuzzy s/t-norm functions with this representation, where the s-norm implements the $\vee$-operator and the t-norm acts as the $\wedge$-operator. This gives the advantage that there are many developed s/t-norms, which could be examined for their impact on EAs solving SAT.

In their experiments, each boolean variable is implemented as a 32-bit gene representing floating point numbers ranging from 0 to 1. Population is initialized with random bits. The authors used one-point-crossover so that crossover is able to split a 32-bit gene. Note that such a split creates a new floating point number around the crossover cut point. Thus, this crossover also explores the search space and is not solely exploiting solution parts of different individuals. For each 32-bit gene mutation flips exactly one of the 32 bits.

An interesting problem arises in their experiments. With increasing size of boolean variables, more and more genes do not converge to a truth value of 0 or 1, but to 0.5. The authors proposed a recursive version of a GA to overcome this problem. The evolutionary search is aborted if a maximum number of generations is achieved. Then all variables which converged to 0 or 1 are fixed and the search starts again and evolves only non-fixed variables. However, there is no more research using fuzzy representation and the authors did not provide a performance comparison, so that the potential of fuzzy representation remains unclear.

Concluding the discussion of representation, there are several different approaches for chromosome encoding, but recent research is mainly using the standard bit-string representation. The reason for this is that the other methods have been proved to be inferior or no comparable results were presented at all.

4.2 Selection

In addition to common selection schemes like roulette wheel or tournament selection, Hao, Lardeux and Saubion [HLS03] extended the selection procedure by means of Hamming distance in bit-string representation. The Hamming distance for two bit-strings is defined as the total number of different bits. Thus two equal bit-strings have a Hamming distance of 0, whereas two bit-strings which differ in every single bit have a Hamming distance of the length of the string.

The selection scheme proposed by Hao et. al. works as follows: A set of

potential chromosomes is selected by fitness proportional selection. Additionally, any two chromosomes in the potential set must have a Hamming distance $\geq d$. This forms the selected set. During search d is automatically adapted, but unfortunately the authors did not provide details about their implementation of this adaption mechanism.

The achieved effect is to guarantee population diversity, if d is chosen big enough. Furthermore different parts of a solution are exchanged faster, just because the bit-strings are chosen under the constraint of Hamming distance.

But this approach has a disadvantage. Suppose a solution space of exactly two possibilities, namely only 0s and only 1s for some N-bit-string. Fitness proportional selection favors strings which have many 1s or 0s. Adding the Hamming distance with sufficiently large d will result in a parent pair, where one is largely composed of 0s and the other of 1s. Performing a crossover on these strings is likely to be not successful for that solution space.

There is a trade-off between classical fitness proportional selection schemes and Hao's extension to them. As aforementioned a classical scheme may encounter the problem of premature convergence (cf. section 3.1 page 9). By utilizing the Hamming distance this procedure increases the population's diversity, thus preventing premature convergence. This is, because crossover is performed on individuals with a large Hamming distance which results in offspring with a large Hamming distance as well. This maintains a diverse population and counters population convergence. However, Hao's extension may not converge to a solution at all, as was argued by example above.

4.3 Fitness Functions

The performance of an EA can be improved by a well designed fitness function, since selection plays an important role on evolutionary computation. Some fitness functions were already presented in the course of this article because fitness functions are closely related to representation. All the following functions use bit-string representation and assume CNF.

4.3.1 SAW

Eiben and van der Hauw [EvdH97] developed an EA which uses stepwise-adaptation-of-weights (SAW) i.e.

$$w_i = w_i + \Delta w,$$

where w_i denotes a weight. The stepwise-adaption is typically implemented with the given assignment and a given value Δw which is applied periodically during the search.

Accordingly Eiben's EA is often referred to SAWing-EA or SAWEA. Essentially SAWEA makes use of the weighing mechanism used by GSAT (cf. section 2.3), which increases the weight of unsatisfied clauses after some time. Thus the SAWEA fitness function of an individual i is defined as:

$$f_{SAW}(i) = w_1 f(i, c_1) + \ldots + w_N f(i, c_N) = \sum_{k=1}^{N} w_k f(i, c_k)$$

where $w_k \geq 1$ denotes a weight and

$$f(i, c_k) = \begin{cases} 1 & \text{if } c_k \text{ is satisfied by i} \\ 0 & \text{otherwise} \end{cases}$$

Initially the weights are set to 1, so that $f_{SAW} = f_{MAXSAT}$. For Δw Eiben et. al. used

$$\Delta w = 1 - f(x^*, c_i),$$

with x^* the current best chromosome in the population. Thus a weight w_i of a clause c_i is increased by 1 if the current best chromosome x^* does not satisfy c_i; Δw increases the weights of unsatisfied clauses. High weights are produced by clauses which are not satisfied for long time periods. The fitness function will prefer satisfied clauses with high weights.

Eiben and van der Hauw adapted the weights every 250 fitness calculations and experimented with different GA settings to find the best GA for SAW. In the end they discovered that a $(1, \lambda^*)$-ES (cf. 3.2.2) performs best. The value of λ^* was determined empirically and differs for varying amounts of boolean variables. Since population has size 1, crossover cannot be performed, but mutation can. The authors introduced the *MutOne* operator, which mutates one randomly chosen gene. All in all $(1, \lambda^*)$ schemata means: take the only possible parent, apply MutOne λ^* times to reproduce at most λ^* different offspring, and select the best of these for the next generation.

The authors also experimented with a $(1 + \lambda)$-ES to avoid precalculating λ^* for different boolean expressions, because λ^* is fine-tuned to special boolean expressions. For this setup the authors claim that the GA behaves insensitive to the setting of λ, if compared to $(1, \lambda)$ (cf. [EvdH97]).

The performance of SAWEA was improved by de Jong and Kosters [JoKo98]. They use $(1, \lambda^*)$-ES variant but supply another mutation operator, which is very similar to local search. This operator works as follows: A chromosome c in offspring is randomly chosen. Furthermore, a set of randomly chosen clauses is generated. If each clause in this set is satisfied by c, then do

nothing. Otherwise pick a random variable of an unsatisfied clause and flip its corresponding bit such that it satisfies the clause. Experimental results reveal that this approach outperforms SAWEA. The authors refer to it as Lamarckian Structural Error Assignment SAW (Lamarckian SEA-SAW)[5].

4.3.2 Refining Functions

Gottlieb and Voss [GoVo98] presented the concept of refining functions. They tried to overcome the problem that f_{MAXSAT} often yields the same value for different bit-strings (cf. section 4.1.1 page 15). The authors suggest to implement a *refining function* r which is used in combination with an ordinary fitness function, e.g. f_{MAXSAT}. This gives a refined fitness function of a chromosome x:

$$f_{ref}(x) = f_f(x) + \alpha r(x),$$

where $\alpha > 0$ is an influence level of the refining function r, $r : \{0,1\}^n \to [0,1)$ and f_f a fitness function. If α is set to zero, f_{ref} behaves like the regular fitness function f_f, whereas if $\alpha \to \infty$, that function is dominated by r. Gottlieb and Voss showed that the influence level α can be easily adapted during the search such that f_f tries to find a solution. But if it gets stuck the influence level α is increased, such that the search is directed to another area (cf. [GoVo98]).

For SAT they propose four different refining functions and choose $f_f = f_{MAXSAT}$. Two of the presented refining functions are counting literal occurrences. Thus, both exploit the problem space of SAT. The third refining function is problem independent, because it only operates on the bit-strings of the population. This method adapts itself during the search by counting how long a bit is set to the same value. After a number of generations, there might be a high valued bit and a flip of that bit is favored. This directs the search to another area of the search space. The fourth refining function is self-adapting as well by utilizing the same mechanism as the third refining function. It additionally exploits domain knowledge of SAT by biasing the search to produce bit-strings which satisfy more clauses.

4.3.3 SAW and Refining Functions

In a later work Gottlieb and Voss [GoVo00] incorporate the SAW mechanism to their refining functions. They define a new refining function r for a

[5] Note that the SEA-SAW operator is similar to the *random walk* strategy, another searching strategy for GSAT ([SeKa93])

chromosome x as:

$$r(x) = \frac{1}{2}\left(1 + \frac{\sum_{i=1}^{N} K(x_i)v_i}{1 + \sum_{i=1}^{N} |v_i|}\right),$$

where $K(0) = -1$, $K(1) = 1$. The authors adapted v_i with SAW by:

$$v_i = v_i - K(x^*) \sum_{k \in U_i(x^*)} w_k,$$

where x^* is the currently fittest individual, $U_i(x^*)$ is the set of unsatisfied clauses in which atom i occurs, and w_k is equivalent to the description of SAW (cf. section 4.3.1). Note that w_k is also adapted during the search. In comparison to SAWEA's weight adapting method, the weights v_i are additionally powered by the cardinality of the unsatisfied clauses containing variable i. The authors also discuss a special case of this refining function using constant weights for $w_i = 1$.

Regarding the success rate[6], this procedure, called RFEA2+, is state-of-the-art for GAs solving SAT. The setting of RFEA2+ is:

- Bit-string encoding
- Population size 4
- RFEA2+ fitness function
- Tournament selection size 2
- Lamarckian SEA-SAW mutation operator
- Steady-state replacement[7]

Note that RFEA2+ utilizes a small population in comparison to De Jong and Spears' first SAT solving GA which uses 100 chromosomes. Also note that no crossover is used, although crossover is regarded as GA's driving search force (cf. section 3.2.3). Performance evaluation are given in section 4.6

[6]Success rate is the percentage of experimental runs where a solution was found.

[7]Steady-state replacement is a scheme which discards worse fitness chromosomes and deletes individuals occurring twice in the population

4.4 Genetic operators

The performance of evolutionary computation can be enhanced if domain knowledge and / or self-adapting methods are used. Some operators of this kind have already been introduced such as the Lamarckian SEA-SAW mutation in section 4.3.1. This section concentrates on research which focuses on crossover and mutation.

4.4.1 Crossover

Classical crossover schemes vary from one-point-crossover to uniform-crossover, which were presented in section 3.1. But these blind crossover operators are commonly considered as inefficient for evolutionary computation for SAT. Park wrote:

> The performance results indicate that for large and difficult 3SAT instances, cross-over fails to be effective, and genetic search is consistently outperformed by simulated annealing. ([Pa95])

Also Gottlieb et. al. argue:

> . . . this indicates that crossover is not dominant, and that local search steps are essential. ([GMR02])

Hence, there are crossovers which exploit domain knowledge by utilizing local search [HLS02]. Supposing CNF and bit-string representation, the authors introduced an improvement function:

$$imp_i = |S_i| - |U_i|,$$

with S_i (U_i) the set of satisfied (unsatisfied) clauses if gene i is flipped.

Through imp_i Hao et. al. defined two groups of 2-parent-crossover functions generating only one offspring. The first group is composed of crossovers using the best value computed by imp_i. Since imp_i is calculated for every gene i, the i with the best improvement value is used for the offspring. The second group of crossovers is designed to preserve satisfied clauses. For unsatisfied clauses imp_i is used to find a good candidate who can satisfy those clauses. An example of a crossover belonging into this group is given below.

The authors evaluated different crossovers from both groups and the results demonstrate that a crossover of the second group performed best. This crossover works as follows: For each clause c_j which is satisfied by both parents X, Y, take those genes v contained in c_j for which $X_v = Y_v$ and c_j

satisfied by v holds. This preserves satisfied clauses. For a clause c_j which is not satisfied by both parents, compute a gene v with a maximal k, where $k = imp_i(X)+imp_i(Y)$, for all genes i occurring in c_j. The inverted v is used for the offspring. This results in the biggest gain of satisfied clauses in the offspring. Finally this crossover initializes the genes which are not processed to random values.

A disadvantage of this biased crossover is higher computational costs owing to the frequent usage of improvement function imp_i.

4.4.2 Mutation

Several mutation operators were already introduced in this paper. For example, mutation operators for non-bit-string representations e.g. mutation using standard deviation (section 4.1.5). For bit-string representation, blind mutation like flip (section 2.3) or MutOne (section 4.3.1) were described. Furthermore Lamarckian SEA-SAW mutation was introduced, which utilizes local search. In addition to these, GASAT [HLS02] implements mutation with a mechanism based on TABU-Search.

GASAT uses crossover as described in the preceding section, and its mutation uses the improvement function imp_i as well. Mutation utilizes a FIFO set T, where T is bounded by a maximum size γ. Genes are placed into T if they are forbidden to be flipped. For each mutation, T is initially empty. The mutation works as follows: Choose gene v with

$$v = \max_i (imp_i(x)) \text{ for all } i \notin T,$$

where x is the chromosome to be mutated. The resulting v is a gene that is allowed to be flipped and will cause the largest improvement if flipped. After this calculation, v is flipped. If the resulting chromosome has a better fitness the flip is kept. Then v is placed into T and eventually displaces an oldest stored gene j. This mutation process is repeated until a maximum number of flips is reached (cf. [HLS02]).

This procedure does not accept lower-fitness solutions so that only fitness improvements ≥ 0 are possible. By establishing T, mutation cannot immediately revert to a former state. Therefore the search is forced to explore a different part of the search space. Nevertheless, the size of γ poses a problem, because a small γ will enable this mutation to revert to a former state too fast. Hence, the search may return to the same local optimum repeatedly. A too big value for γ hinders mutation to run efficiently, because only a few genes are allowed to be altered. Note that this mutation operator uses imp_i

and fitness evaluations frequently. Therfore GASAT's mutation operator is computational expensive.

4.4.3 Local Search

An evolutionary algorithm is also called a *Hybrid-EA*, if it incorporates local search operations besides crossover and / or mutation. A typical example of a local search procedure is GSAT's greedy strategy (cf. 2.3).

Marchiori and Rossi [MaRo99] introduced a flip heuristic for local search in a standard GA with bit-string representation, uniform-crossover and mutation for CNF formulas. Flip heuristic is performed after the genetic operators and computes the increase of satisfied clauses by flipping a gene. Hence, the authors named this GA as FlipGA.

For each offspring, the flip heuristic flips every gene and determines if this flip yields an increase ≥ 0 of satisfied clauses. If there is an increase, the flip is accepted and the increase is added to a zero initialized variable called *gain*. After the whole chromosome is traversed, the search is restarted from the first gene if $gain > 0$, otherwise the local search is halted.

This process was further improved by an adaptation mechanism called ASAP by Rossi, Marchiori and Kok ([RMK00]). In addition to FlipGA a chromosome set T of size K is established, which stores only individuals with the same best fitness. If an individual evolves with a higher fitness value than the current best individuals in T, the set is emptied and the newly generated best fitness individual is put into it. Chromosomes with the same fitness value are added to T and lower fitness individuals are ignored. If the set reaches its maximum capacity K, the contained chromosomes are compared bitwise and all genes are *frozen* except those having the same value in all chromosomes. Frozen genes are not altered by mutation or the flip heuristic. The search is carried on normally until a better fitness individual is discovered. This discovery empties T except the newly discovered chromosome and all genes are allowed to be flipped again[8].

The intention of this adaptation mechanism forces the search to explore another area of the search space, as the search got stuck on a local optimum. Since only non-frozen genes are altered, only equally set variables are allowed to be flipped. Therefore the common characteristics among the best fitness individuals are modified, which may be the reason that the search got stuck on a local optimum.

[8] The genes are not frozen anymore

4.5 Parallel-GA

A parallel implementation of a genetic algorithm solving SAT was done by Folino, Pizzuti and Spezzano [FPS98]. They introduced a method called CG-WSAT. CGWSAT is a cellular GA incorporating local search WSAT [SKC94] which is a variant of GSAT.

CGWSAT uses a 2-dimensional toroidal grid of cells such that every cell has eight neighbors (north, north-east, east, south-east, south, south-west, west, north-west). Folino et. al. used a population size of 320 which corresponds to 320 cells evolving in parallel. Each cell contains exactly one chromosome and can only perform crossover with the fittest of its adjacent neighbors. 2-point crossover is used. Mutation is only applied under some preset probability and follows the WSAT procedure. The fitness function is set to $f = \#Clauses - f_{MAXSAT}$. The authors expect that subpopulation of similar characteristics can evolve due to the neighborhood-restricted crossover operation. Moreover they state that if crossover probability is set to 0 CGWSAT is in fact a parallel version of WSAT.

4.6 Performance Evaluation

Unfortunately there is no publication about performance evaluation which covers all the presented concepts. Furthermore, the discussed articles seldom use comparable test suites of boolean formulas. Therefore, this section is based on the performance evaluation proposed by Gottlieb, Marchiori and Rossi [GMR02] who concentrated on recent EA for SAT developments.

The authors used random 3-SAT boolean formulas of different problem sizes i.e. number of boolean atoms N. A transition phase valued 4.3 is used in these formulas. For detailed specifics of the test suites refer to [GMR02]. Gottlieb et. al. measured *success rate* (SR), i.e. the percentage of runs that achieve a solution, and the *average number of flips to solution* (AFS) as evaluation criteria. Multiple runs were carried out per formula. WSAT, a GSAT variant, is also listed in the performance tables in figure 7 for comparing evolutionary computation with incomplete methods.

The results indicate that evolutionary computation for SAT is competitive against incomplete methods like WSAT. Furthermore, it seems that using domain-knowledge can improve the search, because SAWEA, which uses only blind operators, performs worse. However, note that WSAT typically does not need as many AFS as the other approaches, especially with increasing number of atoms N.

Comparative Result 1 [GMR02]

Algorithm	N = 30		N = 40		N = 50		N = 100	
	SR	AFS	SR	AFS	SR	AFS	SR	AFS
SAWEA	1.00	34015	0.93	53289	0.85	60743	0.72	86631
RFEA2+	1.00	2481	1.00	3081	1.00	7822	0.97	34780
FlipGA	1.00	25490	1.00	17693	1.00	127900	0.87	116653
ASAP	1.00	9550	1.00	8760	1.00	68483	1.00	52276
WSAT	1.00	1631	1.00	3742	1.00	15384	0.80	19680

Comparative Result 2 [GMR02]

Algorithm	N = 40		N = 60		N = 80		N = 100	
	SR	AFS	SR	AFS	SR	AFS	SR	AFS
SAWEA	0.89	35988	0.73	47131	0.52	62859	0.51	69657
RFEA2+	1.00	2951	0.99	19957	0.95	49312	0.79	74459
FlipGA	1.00	14320	1.00	127520	0.73	29957	0.62	20319
ASAP	1.00	16644	1.00	184419	0.72	46942	0.61	34548
WSAT	1.00	5472	0.94	20999	0.72	30168	0.63	21331

Figure 7: Performance Evaluation

5 Conclusion

Section 4 described evolutionary computation for SAT. Many of these recent developments rely on local search, similar to GSAT's greedy stategy. Hence, people wonder:

> Does the EA framework provide a contribution to SAT solvers?

This topic is discussed in the following.

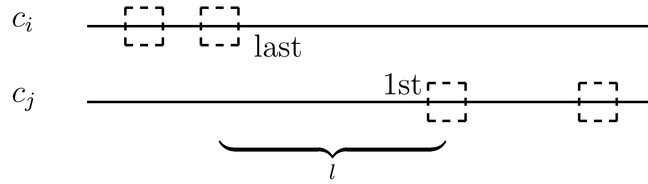
5.1 Criticism on EA solving SAT

The first GA solving SAT by De Jong and Spears [JoSp89] is a classical GA applying population, fitness function and blind genetic operators. However, it is empirically proven that blind genetic operators are not as efficient as applying domain knowledge exploiting operators (cf. section 4.4.1). A reason for this is, that EAs with only blind genetic operators are stochastic search processes since they rely on randomly-chosen crossover and mutation points. To illustrate how the search is influenced by parameter settings and its stochastic search means, consider the following example:

Assume a GA using population size $P = 10$, one-point-crossover applied with probability $p_c = 0.8$, one gene mutation applied with probability $p_m = 0.2$ and a chromosome size of $n = 100$. The focus is on the probability of finding a solution by applying crossover *and* mutation. Therefore, assume that this GA has already run for some time. The current generation contains two chromosomes c_i and c_j such that the application of both genetic operators can lead to a satisfiable truth assignment. For simplicity c_i and c_j have $2k$ different bits which have to be recombined to form a solution. For c_i k bits are in the first half of the chromosome and for c_j k bits are in the second half. Furthermore, mutation has to alter exactly one gene to find the solution e.g. recovering a lost gene. Then we have:

1. Fitness proportional selection selects c_i and c_j with some probability p_s.

2. Crossover must be performed and has to choose the right spot for splitting. This has a probability of $p_c = \frac{8}{10} * \frac{l}{100}$. Consider the following chromosome design of c_i and c_j with $k = 2$:

   l is the number of genes between the *last* needed gene of c_i and the *1st* needed gene of c_j. If $k = 1$ the expectancy is 50. But this value decreases with an increasing of k. Also note that in many cases the $2k$ different bits are not distributed in distinct parts.

3. Performing the right mutation has a probability of $p_m = \frac{2}{10} * \frac{1}{100}$.

The probability of finding a solution directly is:

$$p_s * \frac{8l}{1000} * \frac{2}{1000}$$

This equation depends on parameters l and p_s. For the sake of seeing numbers set $l = 50$ and $p_s = 0.5$. Then the given equation results in a probability of 0.0004 to find a satisfying truth assignment directly by blind crossover and mutation. Note that the chromosome configuration may improve after a unsuccessful execution of either crossover or mutation, e.g. one of the offspring has $k + x$, $x < k$, of the needed genes. However, it is also possible that the chromosome get worse, e.g. crossover does not recombine any of the $2k$

needed genes into one individual, or mutation alters a gene which is crucial for a solution.

Park [Pa95] discussed the general effectiveness of blind genetic operators for SAT. He *decomposed* a bit-string encoded GA into three forms: Crossover-Mutation-Selection, Crossover-Selection, and Crossover-Mutation. He experimented with these three settings and with varying probabilities of genetic operators. Park concluded that:

> The MAX-SAT results ... suggest that for large and difficult problem instances, the building-block hypothesis may be an inadequate characterization, rendering cross-over's role marginal ... ([Pa95])

The building block hypothesis states that parts of a solution are code blocks at different locations in different parents. These code blocks can be assembled by crossover, such that an *improved* solution is built by recombining the partial solutions of the code blocks. Park showed with his experiments that crossover plays no important role for genetic search and inferred that the building block hypothesis is inappropriate for SAT.

Thus, a result of evolutionary computation for SAT research is that blind genetic operators do not provide 'good' means to solve SAT except they can support random jumps in the search space.

To improve the search process, researchers enhanced genetic operators by utilizing domain knowledge or some mechanisms to guide the search e.g. TABU-Search (cf. section 4.4.3). But if such techniques are used, then the question arises if EA's additional procedure contribute to the search.

The main differences between GSAT and an EA using local search are (1) genetic operators, (2) population, and (3) fitness function. In addition different representation possibilities were presented in this paper. However, only the bit-string representation is commonly used since other representations seem to perform less well (cf. section 4.1). Also note, that some of the presented representations, e.g. clausal or path representation, could be combined with a GSAT algorithm[9]. Therefore, other representations despite the bit-string are not considered in the following.

(1) Blind genetic operators are stochastic search means. Experiments have shown that blind crossover is not able to recombine building blocks into

[9] e.g. take clausal representation and change GSAT's greedy strategy to minimize variable inconsistencies

one individual. Furthermore there is also no informed crossover operator having the ability to recombine the population's building blocks into a solution. Hence, mutation is the search's driving force. However, blind mutation is still a random search process. But using an informed mutation operator means applying some GSAT similar strategy since it operates on only one individual. Therefore, genetic operators seem to do not provide a contribution to the search.

(2) Population provides means of parallel search space exploration. This is because crossover, which is the only genetic operator working on multiple chromosomes, is ineffective. Hence, mutation searches the population's individuals so that multiple individuals are searched at the same time / in parallel. Note that population provides means to parallelize other (incomplete) methods, e.g. CGWSAT (cf. section 4.5). However, such a parallelization is similar to a GSAT algorithm running on multiple processors at the same time, because no solution parts are interchanged during the parallel search.

Algorithms like SAWEA or RFEA2+ consider only the best individual in the population for learning the solving difficulty of a clause. To this end global clause weights are applied. Since fitness proportional selection is used, these weights affect not only the solving process of the best individual – from which the weights were derived from – but of all chromosomes. Thus the whole population is forced to the same search space area to which the best individual is lead to. Therefore the population becomes less diverse in the course of time, such that premature convergence may be a consequence.

If global clause weights are bad for the search process of a whole population, then the usage of *local* clause weights for each chromosome are an option. However, this alternative suggests that a population is *only* an implementation of parallel search, because of ineffective crossover and local learning strategies.

Nevertheless, a global strategy like clause weights may accelerate the search process for some search configurations. Consider a diverse population and some clause weights setting derived from a best individual. Then the best individual may find a solution because of the clause weight guidance. But another individual may find a satisfying truth assignment as well, because of the same guidance although it is diverse. It is unclear if such *coincidental* solutions occur often or infrequently.

(3) A main part of EA search lies within the usage of the fitness function, but

note that the fitness proportional selection concept is similar to GSAT's greedy strategy. The fitness function guides the search by selecting individuals which seem promising for crossover to combine different solution parts. Yet, it was argued that crossover does not contribute to the search. However, fitness function is still useful especially for (μ, λ)- and $(\mu + \lambda)$-ES where fitter individuals are chosen for the next generation. The purpose of selecting fitter individuals is to lead the search to a solution, but this is essentially greedy.

Therefore, a fitness function should be designed carefully, such that the greedy selection process is led to a solution. This forms

The Fitness Dilemma:

> Does an increase of fitness lead to the global optimum, i.e. the (only) solution?

Researchers often used f_{MAXSAT} for this guidance, thereby relying on the number of already satisfied clauses. However, a fitness increase calculated by f_{MAXSAT} does not necessarily lead to a SAT solution, e.g. because of variable constraints as described in section 2.3.

Thus, an EA using local search like GSAT and fitness proportional selection is a twofold greedy algorithm. GSAT's local search tries to satisfy a maximum number of clauses and fitness selection chooses fitter chromosomes. Note, that if f_{MAXSAT} is used for the fitness function, both greedy strategies rely on the same fitness landscape. Furthermore, f_{MAXSAT} forms a typical hillclimber problem. Thus, the combination of greedy local search and fitness proportional selection do not address the hillclimber problem since both strategies are based on f_{MAXSAT}. A better approach solving this problem would be to use two different fitness landscapes. Such a combination could 'smooth' the f_{MAXSAT} landscape, such that the global optimum can be found more easily.

But there is also theoretical discussion on the usefulness of f_{MAXSAT}. Rana and Whiteley [RaWh98] do a Walsh analysis of MAXSAT problems and experimentally show the deceptiveness of f_{MAXSAT}. They state:

> This behavior indicates that the problems are not only deceptive, but there are also many equally good regions for a genetic algorithm to explore. If there were biases in the schemata, the genetic algorithm would have converged con-

> sistently to specific regions of the search space for the same problem. Yet the convergence behavior appears to be almost random. ([RaWh98])

However, the performance evaluation of RFEA2+ given in section 4.6 indicates that the combination of both greedy mechanisms is indeed a contribution to incomplete SAT solvers, hence the good results of RFEA2+. But this may be due to the well designed fitness function of RFEA2+. RFEA2+'s refining function enables the search to escape from local optima. Thus, the reason for the good performance is due to the refining function. Therefore RFEA2+'s fitness function may improve the performance of GSAT and if so, it would indicate that greedy fitness proportional selection is not a contribution for SAT search.

The discussion above suggests that the EA framework is not a contribution to SAT solvers. This is because crossover is not able to recombine the building blocks of a solution, population only provides means of parallel searching, and the concept of fitness function is similar to greedy strategies of GSAT variants. Nevertheless the combination of *solution-seeking* and *local-optimum-escaping* functions like RFEA2+ seems promising for further research.

However, the SAT fitness landscape itself is complicated. Michalewicz [Mi95] divides a constrained search space in feasible and infeasible parts as sketched in figure 8.

He says:

> In solving optimization problems we search for a feasible optimum. ([Mi95])

But SAT is essentially a binary function, such that a feasible part of the search space corresponds to a solution. Especially a function with only one possible solution has only one feasible 'dot' in the search space. The transformation of SAT to the optimization problem MAXSAT is not effective, because the MAXSAT fitness landscape is deceptive. However, local optimum escaping techniques as applied in RFEA2+ perform well with f_{MAXSAT}. Nevertheless, crossover and population seem to lack the ability to contribute to SAT search.

5.2 Further Research Possibilities

RFEA2+'s fitness function has to be analyzed in usage with a GSAT algorithm, such that the influence of fitness proportional selection can be verified.

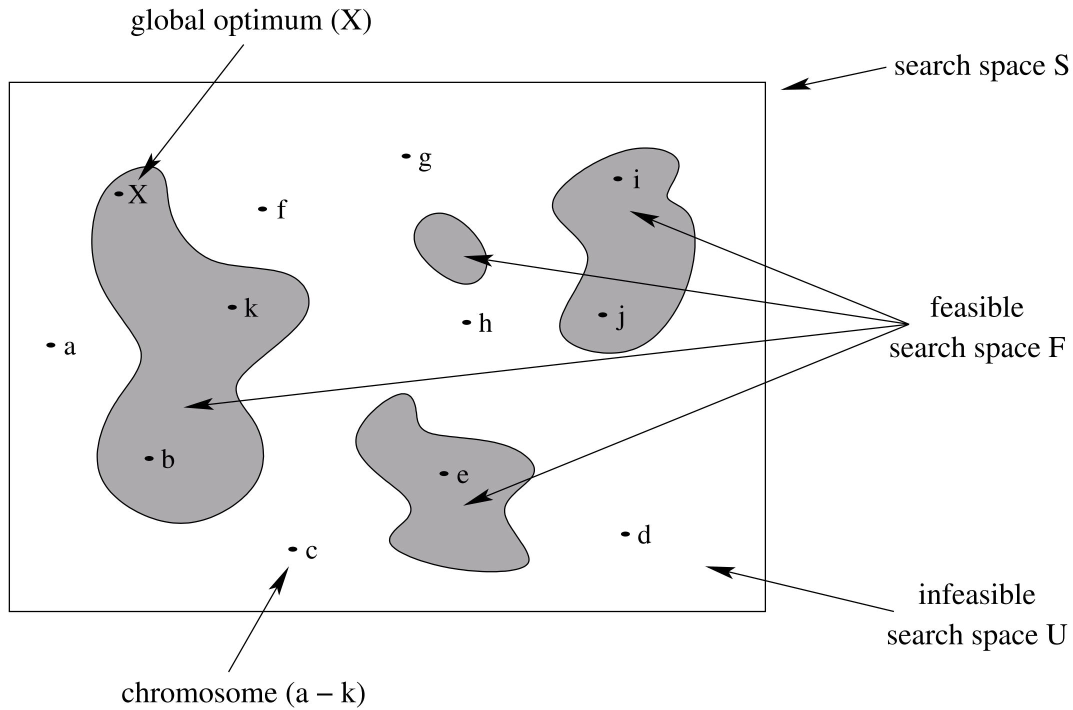


Figure 8: Constrained Search Space

To this end GSAT's greedy strategy has to be replaced by RFEA2+'s fitness function. Performance comparison between both may be able to provide a good indication, if the greedy selection process is a contribution.

For further research it would also be interesting to see if it is possible to learn the solution dependencies (building blocks) of a given formula. This may be possible with an extension of the SAW principle. In addition to clause weights, variable weights may be of importance, perhaps even clause-variable weights.

The commonly used bit-string representation is similar to the Davis-Putnam algorithm, because it focuses on the formula's atoms. Searching with this representation causes unsatisfied clauses, which have to be minimized for a solution. Another approach is to use a representation for clauses like clausal representation or path representation. These may generate atom truth assignment inconsistencies, which have to be minimized to find a solution, as well. However, since hard random SAT instances have a transition phase of 4.3, the maximum number of inconsistencies is smaller than the maximum number of unsatisfied clauses. Therefore, the smaller number of inconsisten-

cies may be a better subject for local search performed by a GSAT variant or an EA.

Furthermore the effects of global clause weights on a population of chromosomes should be analyzed regarding the frequency of solution individuals, which are affected by weights derived from some other individual.

References

[BEV98] T. Bäck, A. Eiben, M. Vink, *A Superior Evolutionary Algorithm for 3–SAT*, In International Conference on Evolutionary Programming, in cooperation with IEEE Neural Networks Council, 1998

[BüLe94] H. Kleine Büning, T. Lettmann, *Aussagenlogik: Deduktion und Algorithmen*, B. G. Teubner, 1994

[Co71] S. Cook, *The complexity of theorem-proving procedures*, In Proceedings of the third Annual ACM Symposium on Theory of Computing, 151–158, 1971

[DaPu62] M. Davis and H. Putnam, *A computing procedure for quantification theory*, Journal of the ACM, 483–497, 1962

[EvdH97] A. Eiben, J. van der Hauw, *Solving 3–SAT by GAs Adapting Constraint Weights*, In Proceedings of The IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence", 1997

[FPS98] G. Folino, C. Pizzuti, G. Spezzano, *Combining cellular genetic algorithms and local search for solving satisfiability problems*, In Proceedings of the Tenth IEEE International Conference on Tools with Artificial Intelligence, 192–198, 1998

[Fr96] J. Frank, *Weighting for Godot: Learning heuristics for GSAT*, In Proceedings of AAAI96, 338–343, 1996

[GoVo97] J. Gottlieb, N. Voss, *Representations, Fitness Functions and Genetic Operators for the Satisfiability Problem*, Artificial Evolution – Third European Conference, Springer, 1997.

[GoVo98] J. Gottlieb, N. Voss, *Improving the Performance of Evolutionary Algorithms for the Satisfiability Problem by Refining Functions*, Parallel Problem Solving from Nature, 755-764, 1998

[GoVo00] J. Gottlieb, N. Voss, *Adaptive Fitness Functions for the Satisfiability Problem*, Proceedings of the 6th International Conference on Parallel Problem Solving from Nature, 621–630, 2000

[GMR02] J. Gottlieb, E. Marchiori, C. Rossi, *Evolutionary Algorithms for the Satisfiability Problem*, In Evolutionary Computation, 35–50, 2002

[HaDo94a] J. Hao, R. Dorne, *A New Population-Based Method for Satisfiability Problems*, European Conference on Artificial Intelligence, 135–139, 1994

[HaDo94b] J. Hao, R. Dorne, *An empirical comparison of two evolutionary methods for satisfiability problems*, Proceedings of IEEE International Conference on Evolutionary Computation, 450–455, 1994

[Ha95] J. Hao, *A Clausal Genetic Representation and its Evolutionary Procedures for Satisfiability Problems*, In Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Ales, 1995

[HLS02] J. Hao, F. Lardeux, F. Saubion, *A Hybrid Genetic Algorithm for the Satisfiability Problem*, First International Workshop on Heuristics, 2002

[HLS03] J. Hao, F. Lardeux, F. Saubion, *Evolutionary Computing for the Satisfiability Problem*, In EvoWorkshops 2003, 258-267, 2003

[JoSp89] K. De Jong, W. Spears, *Using Genetic Algorithms to Solve NP-Complete Problems*, In Proceedings of the Third International Conference on Genetic Algorithms, 124-132, 1989

[JoKo98] M. de Jong, W. Kosters, Solving 3–SAT using adaptive sampling, In Proceedings of the Tenth Dutch/Belgian Artificial Intelligence Conference, 221-228, 1998

[MaRo99] E. Marchiori, C. Rossi, *A Flipping Genetic Algorithm for Hard 3-SAT Problems*, In Proceedings of the Genetic and Evolutionary Computation Conference, 393–400, 1999

[Mi95] Z. Michalewicz, *Heuristic Methods for Evolutionary Computation Techniques*, In Journal of Heuristics, Vol.1, No.2, 177–206, 1995

[MüVo95] H. Mühlenbein and H. Voigt, *Gene pool recombination in genetic algorithms*, In Osman and Kelly [1484], 53–62, 1995

[Pa95] K. Park, *A comparative study of genetic search*, In Proceedings of the Sixth International Conference on Genetic Algorithms, 512-519, 1995

[PSS02] W. Pedrycz, G. Succi, O. Shai, *Genetic-fuzzy approach to the Boolean satisfiability problem*, In IEEE Transactions On Evolutionary Computation, 519-525 (2002)

[RaWh98] S. Rana, D. Whitley, *Genetic Algorithm Behavior in the MAXSAT Domain*, In Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature, 785–794, 1998

[RMK00] C. Rossi, E. Marchiori, J. Kok, An adaptive evolutionary algorithm for the satisfiability problem, In Proceedings of the 2000 ACM symposium on Applied Computing, 463–469, 2000 Year of Publication: 2000

[RoPr99] A. Rogers, A. Pruegel-Bennett, *Genetic Drift in Genetic Algorithm Selection Schemes*, IEEE Transactions on Evolutionary Computation, 1999

[SJBFG93] W. Spears, K. De Jong, T. Bäck, D. Fogel, H. de Garis, *An Overview of Evolutionary Computation*, In Proceedings of the European Conference on Machine Learning, 442–459, 1993

[ScMi97] M. Schoenauer, Z. Michalewicz, *Evolutionary Computation*, In Control and Cybernetics, 307–338, 1997.

[ScSo00] D. Schuurmans, F. Southey, *Local Search Characteristics of Incomplete SAT Procedures*, In Proceedings of the Seventeenth National Conference on Artificial Intelligence, 297–302, 2000

[SeKa93] B. Selman, H. Kautz, *Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems*, In Proceedings of the International Joint Conference on Artificial Intelligence 1993

[SKC94] B. Selman, H. Kautz, B. Cohen, *Noise strategies for local search*, In Proceedings of the 12th National Conference on Artificial Intelligence, 337–343, 1994

[SLM92] B. Selman, H. Levesquem D. Mitchell *A New Method for Solving Hard Satisfiability Problems*, In Proceedings of the Tenth National Conference on Artificial Intelligence, 440-446, 1992

[SML96] B. Selman, D. Mitchell, H. Levesque, *Generating Hard Satisfiability Problems*, In Artificial Intelligence 81, 1996

[Sp95] W. Spears, *Adapting Crossover in Evolutionary Algorithms*, In Proceedings of the Fourth Annual Conference on Evolutionary Programming, 367–384, 1995

[Wi01] D. Whitley, *An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls*, Journal of Information and Software Technology, 817–831, 2001