Abstract

In this paper we propose a novel heuristic based genetic algorithm for solving the satisfiability problem. The idea is to act repeatedly on a population of candidate solutions: at each iteration, first a simple local search procedure is applied to each element of the population; next the genetic operators (selection, recombination and mutation) are applied to the resulting population.

Extensive experiments are performed on benchmark instances from the literature. The results of the experiments are rather satisfactory, and indicate that our algorithm outperforms three recent algorithms based on evolutionary computation that have been reported to be rather effective for solving hard 3-SAT problems.

1 Introduction

The satisfiability problem (SAT) is a paradigmatic NP-complete problem (Garey and Johnson, 1979) with relevant practical applications, like consistency check in expert system knowledge bases (Nguyen et al., 1985), asynchronous circuit synthesis (Gu and Puri, 1995; Puri and Gu, 1996), etc. The SAT problem can be formulated as follows: given a set of clauses $C_1, \ldots, C_m$ on the boolean variables $x_1, \ldots, x_n$, determine if there is an instantiation for the variables such that the formula $C_1 \land \ldots \land C_m$ evaluates to true. A clause is a disjunction of literals, e.g., $x_1 \lor \overline{x}_2 \lor x_3$, where a literal is a boolean variable $x$ or its negation $\overline{x}$. A boolean variable is a variable which can assume only the values true or false. If each $C_i$ contains exactly $k$ distinct literals, then the problem belongs to the $k$-SAT class.

Existing methods for SAT can be roughly classified into two categories: complete and incomplete methods. Efficient examples of the first category include the approaches based on the Davis-Putnam algorithm (e.g., (Gu et al., 1997)). Incomplete methods include approaches based on local search (see the survey (Battiti and Protasi, 1998)) as well as approaches based on evolutionary computation (e.g., (Bäck et al., 1998; Eiben and van der Hauw, 1997; Fleurant and Ferland, 1996; Hao, 1995)). Genetic algorithms for SAT employ heuristic information into the fitness function and/or into the GA operators (selection, crossover, and mutation). For instance, in the recent paper (Gottlieb and Voss, 1998), an adaptive fitness function that uses information about the structure of SAT instances is considered.

The aim of this paper is to show how a rather successful GA-based algorithm for hard 3-SAT problems can be designed by combining a naive GA with a simple local search algorithm. The idea of incorporating local search into genetic algorithms is not new (Kolen and Pesch, 1994; Mühlenbein et al., 1988), and it has been successfully applied to different combinatorial optimization problems. The approach we employ is also known as genetic local search (GLS) (or more in general, memetic search) (Merz and Freisleben, 1997; Moscato, 1989). We design a novel GLS algorithm for solving hard 3-SAT problems which consists of the repeated application of the following two steps to a population of candidate solutions. First, a simple local search procedure is applied to each candidate solution. Next blind GA operators (selection, crossover, mutation and replacement) are applied to the resulting population. Extensive experiments conducted on benchmark instances from the literature support the effectiveness of this approach for solving hard SAT problems.

The rest of the paper is organized as follows. The next section describes the genetic local search scheme
BEGIN
 t := 0;
 initialize P(t);
 (*) apply local search to P(t);
 evaluate P(t);
 WHILE (NOT termination-condition) DO
 BEGIN
 t := t+1;
 WHILE (|P(t)| < |P(t-1)|) DO
 BEGIN
 select parents from P(t-1);
 recombine parents
 mutate children
 (*) apply local search to children
 insert children into P(t)
 END
 END
 END

Figure 1: GLS scheme

and introduces the GLS algorithm for solving 3-SAT. Section 3 contains an experimental analysis of this algorithm. In Section 4 we analyze the effect of the heuristic and of the genetic operators on the performance of the algorithm. Finally, Section 5 contains some conclusive remarks on the contribution of the paper.

2 Genetic Local Search for 3-SAT

Genetic local search (GLS) is a population based iterative search scheme for combinatorial optimization problems. Roughly, it consists of the application of genetic operators to a population of local optima produced by a local search procedure. The process is iterated until either a solution is generated or a maximal number of generations is reached. Genetic local search has been applied with success to various paradigmatic combinatorial optimization problems (e.g., (Marchiori, 1998; Merz and Freisleben, 1997)).

The GLS scheme that is used in our algorithm is illustrated in Figure 1, where |P(t)| denotes the cardinality of the multi-set P(t). The idea is to combine a simple GA with a local search procedure, where the GA is used to explore the search space, while the local search procedure is mainly responsible for the exploitation.

In order to design our GLS algorithm for 3-SAT we have to design the local search and the genetic features that are specific to this problem. We call the resulting GLS algorithm FlipGA (Flip based Genetic Algorithm).

2.1 The Flip Heuristic

We use a problem representation that is usually employed for this problem. A candidate solution is a string of bits having length equal to the number of variables of the considered problem instance. A 0/1 in the i-th entry of the string indicates that the i-th variable has been instantiated to false/true, respectively.

BEGIN
 generate random permutation S of [1..n_vars]
 improve=1;
 WHILE (improve > 0) DO
 BEGIN
 i=1;
 WHILE (i < n_vars) DO
 BEGIN
 flip S(i)-th gene;
 compute gain of flip;
 IF (gain >= 0)
 BEGIN
 accept flip;
 improve=improve+gain;
 END
 i=i+1;
 END
 END
 END

Figure 2: Flip Heuristic

The local search algorithm used in FlipGA takes as input a candidate solution and yields a candidate solution which cannot be improved by flipping any entry: the relative procedure, called Flip Heuristic (FH) is illustrated in Figure 2, where n_vars denotes the number of variables of the considered problem instance. The inner loop of the Flip Heuristic considers each variable: the (value of that) variable is flipped, and the flip is accepted if the gain, that is, the number of clauses that are satisfied after the flip minus the number of clauses that are satisfied before the flip, is greater or equal than zero (if the test gain>0 is satisfied). When all the variables have been considered (when i=n_vars), the process is repeated if the number of clauses satisfied by the obtained chromosome is increased (if the test improve>0 is satisfied). The order in which the variables are considered is random, and it is implemented by means of a random permutation of the indices [1...n_vars].

In (Koutsoupias and Papadimitriou, 1992) it has been shown that the greedy variant of the Flip Heuristic (where the test (gain≥0) is replaced with (gain>0))
is rather effective for solving 3-SAT instances where the number of clauses is $O(n_{vars}^2 \cdot n_{vars})$. However, the most difficult 3-SAT problems have been identified as those whose ratio of the number of clauses to the number of variables is approximately equal to 4.3 (Mitchell et al., 1992). Instances satisfying this property are said to lay in the phase transition.

It is useful to analyze the computational cost of one iteration of the inner loop in the Flip Heuristic: the main computational effort is caused by the calculation of the gain yield by a flip. In order to calculate the gain of a flip, one has to examine those clauses containing the flipped variable or its negation. For random problem instances, it can be shown (cf., e.g. (Spears, 1996)) that the number of clauses that have to be processed for computing the gain of a flip is (on the average) $3 \cdot R$, where $R$ denotes the ratio of the number of clauses to the number of variables.

The Flip Heuristic is used in the local search steps of our GLS algorithm (steps labeled by (*) in Figure 1).

### 2.2 The Genetic Algorithm

The main features of the genetic algorithm component of FlipGA can be summarized as follows:

**Representation** A chromosome is a candidate solution.

**Fitness** The fitness of a chromosome is equal to the number of clauses that are satisfied by the truth assignment represented by the chromosome.

**GA type** Generational genetic algorithm (see Figure 1) with elitist selection mechanism which copies the best two individuals of a population to the population of the next generation (Jong, 1975).

**Genetic operators** We use the following two GA reproduction operators:

- **Crossover** (always applied): uniform (Syswerda, 1989).
- **Mutation**: (applied to each chromosome with probability 0.9): for every gene, with probability 0.5 flip its value.

We use a rather small population, consisting of 10 individuals, because, according to our computational experience, larger populations affect the efficiency of our algorithm and do not bring effective improvements on the quality of the results.

It is worth observing that the genetic operators we use are blind, that is, they do not use information about the structure of the 3-SAT problem, and they perform all choices in a random way. This is counterbalanced by the Flip Heuristic, which transforms chromosomes into maximal partial solutions.

### 3 Experimental Results

In order to assess the effectiveness of FlipGA, we perform experiments on a number of benchmark instances from the literature.

We consider the problem instances used for testing two GA-based algorithms that have been reported to be successful in solving hard 3-SAT instances ((Bäck et al., 1998; Gottlieb and Voss, 1998)). Both these algorithms incorporate adaptive mechanisms in the fitness function in order to bias the search towards better individuals. The best of the algorithms introduced in (Bäck et al., 1998), here called SAW, uses the $(1, A^*)$ selection strategy, a mutation operator changing exactly one bit, and the so-called SAW-ing mechanism for adapting the fitness function. The algorithms introduced in (Gottlieb and Voss, 1998), here called RFGA, use a heuristic mutation operator, no crossover, and an adaptive fitness function which incorporates information on the structure of the SAT problem. Comparative results on these benchmark instances are reported in Tables 1 and 2.

Moreover, we perform experiments on the problem instances used to test a heuristic algorithm introduced in (de Jong and Kosters, 1998), based on a hybrid method mixing features from evolutionary and neural computation, called Lamarckian SEA-SAW. The results of the experiments are reported in Figures 3 and 4.

Finally, we test FlipGA on other 3-SAT instances including instances from the DIMACS Implementation Challenge (Johnson and (Eds.), 1996). However we cannot compare our results with the ones of any of the above mentioned algorithms, since the latter have not been tested on these instances. The results of the experiments are reported in Table 4.

Before discussing the results, we summarize the characteristics of the benchmark instances considered in the experiments. We consider a total of 798 instances, all satisfiable, belonging to the following classes:

- Test suite A (experiments in Tables 1, 2, 3, Figures 3, 4) consists of 662 instances. These are random 3-SAT instances generated using the problem generator written by Allen van Gelder and collected by Jens Gottlieb (see http://www.in.tu-clausthal.de/~gottlieb/benchmarks/3sat) for comparison purposes. These include the 12 instances named
- Test suite C (experiments in Table 4) consists of 36 instances. These are 3-SAT instances from E. Miyano (see ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf). The families are named aim n-x-y where n=50,100,200 is the number of variables and x.y is the clause/variable ratio, including 2.0, 3.4, 6.0. These are artificially generated satisfiable instances having exactly one solution.

The algorithms performance is evaluated by two measures. The Success Rate (SR) is the percentage of instances where a solution has been found. The Average number of Evaluations to Solution (AES) is the average number of fitness evaluations, i.e., the number of newly generated candidate solutions in successful runs. However, since our algorithm applies local search to each chromosome in every generation, we have to take into account also the computational effort of the local search.

We have already seen in Section 2.1 that for randomly generated instances the cost of a flip in terms of the number of clauses that have to be processed in order to compute the gain of a flip is (on the average) $3 \times R$, where $R$ denotes the ratio $n_{\text{clauses}}/n_{\text{vars}}$ of the number of clauses to the number of variables. Thus in order to evaluate a chromosome after a flip, we have to process (on the average) $3 \times R$ clauses.

On the other hand, observe that the cost of a fitness evaluation in terms of the number of clauses that have to be processed in order to compute its value is equal to $n_{\text{clauses}}$. Thus the cost of the evaluation of the fitness after a flip relative to the cost of the fitness of a chromosome is (on the average) equal to $3/R$. This allows one to compute the effort due to the application of the local search, that we denote by FES (Flip cost in terms of number of fitness Evaluations to Solution). If we denote by $n_{\text{flips}}$ the total number of flips computed during a successful execution of the algorithm, then we have

$$\text{FES} = n_{\text{flips}} \times 3/n_{\text{vars}}.$$ 

Therefore in each figure we have reported between brackets in the column labeled AFES the Average FES's in successful runs.

This allows one to perform a fairer comparison of FlipGA with the other two GA-based algorithms, by considering the sum of AFES and AES.

Algorithms of Tables 1-3 terminate if a solution is

<table>
<thead>
<tr>
<th>Inst</th>
<th>n</th>
<th>m</th>
<th>Alg</th>
<th>SR</th>
<th>AES (AFES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>129</td>
<td>FlipGA</td>
<td>1</td>
<td>10 (110)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>253</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>754</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>129</td>
<td>FlipGA</td>
<td>1</td>
<td>180 (1781)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>14370</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>88776</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>129</td>
<td>FlipGA</td>
<td>1</td>
<td>72 (712)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>494</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>12516</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>172</td>
<td>FlipGA</td>
<td>1</td>
<td>11 (178)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>549</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>3668</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>172</td>
<td>FlipGA</td>
<td>1</td>
<td>10 (155)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>1609</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
<td>172</td>
<td>FlipGA</td>
<td>1</td>
<td>104 (1514)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>7648</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>0.78</td>
<td>154590</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
<td>215</td>
<td>FlipGA</td>
<td>1</td>
<td>10 (209)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>480</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>2837</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>215</td>
<td>FlipGA</td>
<td>1</td>
<td>20.40 (415)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>8991</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>8728</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
<td>215</td>
<td>FlipGA</td>
<td>1</td>
<td>570 (11103)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>0.92</td>
<td>85005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>0.54</td>
<td>170664</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>430</td>
<td>FlipGA</td>
<td>0.54</td>
<td>2696 (129657)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>0.16</td>
<td>127885</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>178520</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>430</td>
<td>FlipGA</td>
<td>1</td>
<td>33 (1570)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>18324</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>43767</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>430</td>
<td>FlipGA</td>
<td>1</td>
<td>32 (1564)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RFGA</td>
<td>1</td>
<td>15816</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAW</td>
<td>1</td>
<td>37605</td>
</tr>
</tbody>
</table>

Table 1: Test suite A, part 1

1 to 12 (part 1), and classes named n50, n75 and n100 (part 2). Each class contains 50 instances for the corresponding number of variables. Part 3 contains classes used in (de Jong and Kosters, 1998), called dejong n, with $n=20,40,60,80,100$ being the number of variables. Each class contains 100 instances. All these instances are in the phase transition, and thus have $4.3 \times n_{\text{clauses}}$.

- Test suite B (experiments Table 4) consists of 100 instances. These are uniform distributed, randomly generated 3-SAT instances that are forced to be satisfiable. These instances can be retrieved at the SATLIB WEB site (see http://aida.informatik.th-darmstadt.de/~hoos/SATLIB). We run experiments only on the largest set, with 200 variables and 860 clauses in each instance. All these instances are in the phase transition region. In Table 4 this set of problems is denoted by uf860.
found or the limit of 300000 generated candidate solutions is reached, except for the entry labeled (conv) where convergence has been used as the only stopping criterion. This latter stopping criterion is used also for the experiments reported in Table 4. The results in Tables 1-3 are based on 50 independent runs for each instance, while the results in Table 4 are based on 10 independent runs.

Table 3: Results of FlipGA on Test suite A, part 2, families n75 and n100

The results of the experiments reported in Tables 1, 2\(^1\) indicate that FlipGA outperforms the other two families of algorithms, both in terms of success rate as well as average number of fitness evaluations. In particular, on instance 10, where SAW and RFGA have difficulties in finding a solution, FlipGA reaches the global optimum in each of the 50 runs. It is worth noting that in (Gottlieb and Voss, 1998) four variants of the algorithm where proposed. In table 1 we have reported for each instance the best result of all the algorithms.

Figures 3 and 4 are taken from (de Jong and Kosters, 1998), where we have added the curve describing the behaviour of FlipGA on the same problem instances. All the algorithms terminate if a solution is found or the limit of 300000 generated candidate solutions is reached. Our results are based on the average of 5 independent runs per instance, while the results of the other algorithms are based on the average of 3 runs per instance.

In Figure 3 observe that on larger instances the success rate of our algorithm is much higher than the one of the other algorithms. This indicates that our algorithm has a better scale-up behavior on these instances.

Concerning the average number of evaluations required to find a solution, we have reported for FlipGA the AES+AFES which we have seen provides a fair measure of the (average) cost of a successful run in terms of fitness evaluations. One can see that FlipGA seems to have a scale-up behaviour which exhibits a worse growth than the one of the other algorithms: this result has to be read carefully. In fact, the success rate of the other algorithms becomes rather low when instances having many variables are considered, while the success rate for FlipGA remains rather high. As a consequence, since the computation of AES takes into account only the number of successful runs, the AES for FlipGA is based on much more runs than the AES for the other algorithms. This renders difficult to compare the results.

The performance of FlipGA on the other considered benchmark instances (Table 4) is in general satisfactory. On the uf860 instances, FlipGA is able to find a
Table 4: Results of FlipGA on Test Suites B and C

<table>
<thead>
<tr>
<th>Instance</th>
<th>n</th>
<th>m</th>
<th>SR</th>
<th>AES (AFES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aim50-2.0</td>
<td>50</td>
<td>100</td>
<td>1</td>
<td>46077 (458295)</td>
</tr>
<tr>
<td>aim50-3.4</td>
<td>50</td>
<td>170</td>
<td>1</td>
<td>36 (439)</td>
</tr>
<tr>
<td>aim50-6.0</td>
<td>50</td>
<td>300</td>
<td>1</td>
<td>10 (139)</td>
</tr>
<tr>
<td>aim100-2.0</td>
<td>100</td>
<td>200</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>aim100-3.4</td>
<td>100</td>
<td>340</td>
<td>1</td>
<td>138 (2054)</td>
</tr>
<tr>
<td>aim100-6.0</td>
<td>100</td>
<td>600</td>
<td>1</td>
<td>10 (185)</td>
</tr>
<tr>
<td>aim200-2.0</td>
<td>200</td>
<td>400</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>aim200-3.4</td>
<td>200</td>
<td>680</td>
<td>1</td>
<td>917 (17429)</td>
</tr>
<tr>
<td>aim200-6.0</td>
<td>200</td>
<td>1200</td>
<td>1</td>
<td>13 (283)</td>
</tr>
<tr>
<td>ufs60</td>
<td>200</td>
<td>860</td>
<td>0.78</td>
<td>2393 (42664)</td>
</tr>
</tbody>
</table>

The results of these experiments are based on 50 runs within a limit of 150000 generations (corresponding to 1200010 fitness evaluations).

solution in 78% of the runs. On the aim families (test suite C) FlipGA gives rather good results, except on instances of the aim-50-2_0-yesl* families. In fact, the effort needed by FlipGA to find a solution for instances in aim-50-2_0-yesl* is rather heavy, while FlipGA is not able to find a solution within an hour of CPU time on the instances of the families aim-100-2_0-yesl* and aim-200-2_0-yesl*. Notice that these instances are solved in a few seconds by other effective heuristic algorithms based on local search (Johnson and (Eds.), 1996). It is not clear why our algorithm fails on this specific family, while it does perform very well on other aim families which have been reported to be hard to solve: it seems that FlipGA is not effective on problems where the ratio n. clauses/n. vars is between 1.5 and 2.

4 Discussion

In this section we analyze how FH and the genetic operators affect the performance of FlipGA.

4.1 Role of Flip Heuristic

It is interesting to analyze the role of the local search procedure on the performance of FlipGA. As one would expect, the performance of the simple GA, that is FlipGA without local search, is rather poor. For instance, on relatively easy instances like instance 1 of Table 1, the GA has a success rate SR = 0.78 and average number of fitness evaluations AES = 245627. On harder instances, like instance 6 and 10, the GA is never able to find a solution, getting stuck on partial solutions of scarce quality2.

Let us now analyze the performance of the Flip Heuristic alone. The multi-start version of FH outperforms the simple GA, yet its performance remains rather inferior than the one of FlipGA. For instance, on instance 1 of Table 1, the heuristic has a success rate SR = 1 and AES = 766. On instance 6, the heuristic has a success rate SR = 0.68, and AES = 830. Finally, on instance number 10, FH is never able to find a solution3.

A peculiar characteristic of the Flip Heuristic is that it flips the value of a variable also when the fitness of the chromosome does not change. It is interesting to analyze the performance of FlipGA when the application of this step is restricted by means of a so-called side step probability (ss_prob). This amounts to replacing the IF statement of FH with the one in Figure 6, where rand denotes a random real number in [0,1], and ss_prob is a real number in [0,1] denoting the side step probability. We call the resulting heuristic Restricted Flip Heuristic (RFH). Notice that the original Flip Heuristic can be obtained by setting ss_prob = 1.

Figure 5 illustrates the typical behaviour of FlipGA with RFH for various values of the side step probability, on two relatively easy instances of Figure 1. On these instances, the change of side step probability does not affect the success rate, which remains equal to 1, while the number of generations needed to find a solution increases when the side step probability decreases. On harder instances, like instance 10, the side step probability affects also the success rate, and the best SR is obtained by setting ss_prob = 1.

3The results of these experiments are based on 36000 independent runs, corresponding to the pool size (10) times the number of runs used in our experiments (50) times the maximum number of generations (72) needed by FlipGA to find a solution on the 12 instances of Table 1.
IF (gain>0)
BEGIN
accept flip;
improve=improve+gain;
END
ELSE
BEGIN
IF (rand <= ss_prob)
BEGIN
accept flip;
improve=improve+gain;
END

Figure 6: IF statement with restricted side step

4.2 Role of Genetic Operators

We turn now to the analysis of the performance of FlipGA when the mutation and crossover rate are modified.

- **Mutation** Figure 7 illustrates the typical behaviour of FlipGA when the mutation rate is changed, in terms of the average over 10 runs of the number of generations needed to find a solution. While the success rate remains equal to 1, the experiments show that a high mutation rate causes the algorithm to converge more rapidly to a solution. This seems to indicate that the search for a solution does benefit from a heavy exploration which searches the neighbourhood of a chromosome obtained by flipping half of its bits.

- **Crossover** The behaviour of FlipGA when the crossover rate is modified is not easy to analyze. For example, on small instances of Figure 1 it seems that the crossover does not play any role in the search, while for other instances, like e.g. instance 10, the crossover seems to be crucial for finding a solution, since only few solutions are found when a lower crossover rate is used. Thus our choice of crossover rate 1 is only justified by the improvements obtained on some of the hard instances considered. However, we cannot claim that a high crossover rate does in general improve the performance of FlipGA.

5 Conclusion

In this paper we have introduced an effective GA based algorithm for solving 3-SAT. The main novelty with respect to previous work on this subject is the use of a separate local search algorithm for improving chromosomes. Previous GA approaches for solving 3-SAT have used problem dependent fitness functions together with adaptive mechanism, for identifying difficult clauses and bias the search consequently. In contrast, we use a naive fitness function which describes the number of clauses a chromosome satisfies. We can use such a simple fitness function because in FlipGA at each iteration the population consists of (maximal) partial solutions, thanks to the application of the local search to the chromosomes. In this way, the search pressure determined by the fitness function is exclusively directed towards large partial solutions, while the application of the heuristic together with the genetic operators are responsible for improving the quality of the chromosomes.

This could partly explain why our simple heuristic based genetic algorithm outperforms the other two GA based approaches for 3-SAT: it combines the search for a large partial solution and the search for a partial solution in a neat way, by incorporating the search for 'maximum' and 'partial solution' into the fitness function and the heuristic procedure, respectively.

The choice of the local search procedure to be incorporated in the GA seems to be crucial for the effectiveness of the resulting algorithm: the Flip Heuristic works well on instances in the phase transition, that is where the ratio n clauses/n vars~ 4.3. However, the heuristic seems to have problems to direct the search towards the solution on hard instances having a single solution for which the ratio is between 1.5 and 2. We are actually studying how to extend the local search procedure in order to be able to deal also with such instances.

Finally, an interesting topic that we intend to investigate in the future is the extension of FlipGA for detecting also inconsistency.
References


