Solving Satisfiability Problems with Genetic Algorithms

Stefan Harmeling
Stanford University
harmeli@cs.stanford.edu

March 9, 2000

Abstract

We show how to solve hard 3-SAT problems using genetic algorithms. Furthermore, we explore other genetic operators that may be useful to tackle 3-SAT problems, and discuss their pros and cons.

1 Introduction

Given a propositional formula like

$$(p_1 \vee p_2 \vee \neg p_3) \wedge (\neg p_1 \vee p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee \neg p_3 \vee p_4)$$

with propositional variables p_1, p_2, p_3, p_4 we ask whether there is an assignment of truth values to these variables such that the truth value of the given formula is true, and if so, what is the fulfilling assignment. In our example the assignment

$$p_1 = false,\ p_2 = true,\ p_3 = true,\ p_4 = true$$

makes the given formula true. This problem is the so-called satisfiability problem (SAT). In this paper we consider only formulas in conjunctive normal form[1], where each clause has exactly three literals. These problems are called the 3-SAT problems.

Why is the 3-SAT problem interesting? A number of other problems can be reformulated into 3-SAT, for instance the travelling salesperson problem, the n-queens problem, a variety of planning problems and many more. Furthermore

[1] Recall that a formula is in conjunctive normal form if it is a conjunction of clauses, each of which is a disjunction of literals. Literals are either propositional variables or negated propositional variables.

it is very difficult to solve: Cook showed in 1971 that the 3-SAT problem is NP-hard (Cook 1971). Therefore, we can not expect to solve 3-SAT problems in general. There will be always problems where our approach using genetic algorithms will take an exponential amount of time to find a solution.

2 3-SAT as a Genetic Algorithm Problem

The input of our program is a formula in conjunctive normal form. We evolve a population of variable assignments until we find an assignment that makes the formula true. The assignments are encoded as strings of bits, the length of which is the number of variables; for instance our initial example from the previous section becomes the string of bits:

0111

Obviously, there is a one-to-one relationship between bit strings of length n and truth assignments to n variables. Therefore the search space that we have to search has exactly 2^n elements.

What is the fitness of an individual? We count the number of clauses that are fulfilled by the corresponding variable assignment. The individual that makes the most clauses true is the best one.

3 Genetic Operators

As usual, the first generation consists of randomly created individuums. For the next generation individuums are randomly picked dependent on their performance, that is the ones that got a better fitness are more likely to appear in the new generation. Then we do crossover between two individuals of the chosen ones. We use three different types of crossover:

1. A cutting point is randomly chosen and the two individuals exchange the parts from that point to the end. This is the usual way to do crossover, e.g. the two individuals

   0111 1010

   with cutting point three become:

   011|0 101|1

2. Since the numbering of the variables does not reflect any structure in the formula, it makes sense to consider arbitrary recombinations, that is we choose randomly a string of bits that we will call a map, and recombine

in the following way. All the bits in the two individuums where the map has a one will stay in the same position, and the bits where the map a zero will get exchanged. For instance the map 1001 defines two groups of positions: the two outmost bits and the two middle bits. It changes the two individuals

0111 1010

to

0011 1110

In this example the two bits in the middle are exchanged, that are exactly the positions where the map has zeroes, and the outside bits stay the same, that are exactly the position where the map has ones.

3. Usually, the power of crossover is, that substructures that are already working well don't get destroyed easily. What are in our case substructures? Variables that are in the same clauses can be seen as substructures. Since variables directly correspond to positions in those maps from the previous item, we should try to choose only maps, where it is very likely that variables that are in the same clause end up in the same group. To do this we define a graph where each node corresponds to a variable. We have an edge between two variables if and only if there is a clause in which those two variables both appear. Then we evaluate using an all-shortest-pair-algorithm a matrix that contains the shortest distances between two nodes, that is between two variables. This matrix gets computed once immediately after the formula is loaded. This matrix is used to choose maps in which two variables appear very likely in the same group if they are close in the graph. We do this by first randomly generating a map, that is just a string of bits as explained before. Then we choose randomly a position in the map and evaluate the sum *pros* of all distances of the variable that corresponds to that chosen position to all other variables that are in the same group according to the current map and the sum *cons* of all distances of that variable to all other variables that are not in the same group. If $cons > pros$ holds the chosen position in the map flips. We repeat checking and flipping dependent on the *convergence rate*. For instance if we have 100 variables, that means automatically that the map is 100 bits long, then a convergence rate of 0.5 means, that we randomly choose and possibly flip 50 times. Having determined a map we simply perform the crossover as explained in the previous item.

We will show results for the three different crossover methods in the next section. Another feature of our genetic operators is that the best individual of a generation will survive the crossover process for sure, unless explicitely all individuals are used for crossover. This has been proven very useful during our experiments.

How do we avoid to get stuck in local maxima? As usual we implemented mutation as a genetic operator. Additionally we add at each generation a small

number of completely new individuums. Hereby, we try to keep our population fresh. We call this feature *out-of-the-blue*. The number of new individuums is called *out-of-the-blue-rate*.

4 Experiments and Results

Randomly generated 3-SAT problems are not necessarily hard to solve. As Selman et al. have shown hard problems are those where the variables to clauses ratios is about 4.3 (Selman 1996). Our formulas have 20 variables and 86 clauses. To get hard problems we used the algorithm mentioned in their paper called random K-SAT. It randomly generated 86 clauses of length 3. Each clause is produced by randomly chosing three variables from the set of 20 variables and negating it dependent on a fair coin flip. Then we used the Davis-Putnam procedure (Selman 1996) to sort out the unsatisfiable problems. We ended up with 71 satisfiable problem that we used for all our experiments.

The first question we had is: can we solve any of those hard problems using genetic algorithms? After trying various different parameter settings we solved 59 of those 71 problems using the following parameters:

- Each population has 200 individuums.
- We run maximally 10000 generations.
- We use standard crossover with crossover rate 0.9.
- The mutation rate is 0.0001.
- We produce in each generation one completely new individuum (out of the blue – as described in the previous section).

The details can also be seen in the following tableau:

To get in the following comparable results we set the crossover rate constant to 0.9 and the mutation rate constant to 0.0001 and the maximum number of generation constant to 10000.

For each of the following series we let vary exactly one parameter. For each value of the changing parameter we computed for each of the 71 problems the number of generations needed to solve it. The median of those numbers gives us a valid measure how quick we converged. The average of those numbers would have been less informative because the very large outlayers (their number of generations is 10000) resulting from problems that are not solved after 10000 generations would distort the whole picture. The following tables show the value of the just described median in relation to one changing parameter at a time:

Objective:	To solve hard 3-SAT problems.
Representation Scheme	Structure: fixed length string; $gene_i$ = truth value of i-th variable $K = 2$ L = number of variables Mapping: $0 = false$, $1 = true$
Fitness cases:	Only one.
Fitness:	The number of clauses fulfilled so far.
Parameters:	$M = 200$
Termination criteria:	all clauses fulfilled or more than 10000 generations
Result designation:	solved or not solved.

Figure 1: Tableau for solving 3-SAT problems

- How many individuals do we need per generation?

50	100	150	200	250	300	individuals per generation
348	176	128	81	49	58	median

The table suggests that we should have at least 250 individuals per generation. We used in the subsequent experiments only 200 to decrease the computation time.

- How useful is the out-of-the-blue feature?

0	1	2	3	4	5	10	out-of-the-blue-rate
46	48	39	43	47	65	50	median

After looking at this table we better ask: Is out-of-the-blue a useful feature at all? Unfortunately, there is not a big difference in the values between not using this feature (out-of-the-blue-rate = 0) and using it (out-of-the-blue-rate < 0). Also for positive out-of-the-blue-rate there is no outstanding setting identifiable. Though the sample size is small, we conclude that the out-of-the-blue feature does not increase the speed of convergence.

- How do the different crossover strategies compare?

1	2	3	crossover-type
81	48	48	median

We see that choosing a more flexible crossover policy is superior over just cutting the bit strings into two pieces. As mentioned before for the 3-SAT problem the individuums have no structures, that is reflected by the arrangement of the bits in the corresponding string. A more flexible crossover scheme like the second one does not destroy more patterns than a simple cut. On the contrary, it provides a more powerful way to search

the space of possible solutions. What about the third strategy? What could explain that it didn't do any better than the second? For the hard problems generated by random K-SAT the graph, that was defined in the section in which we explain the different strategies, has a special form. We evaluated the distance matrix for a number of different problems. A typical one of those is the following:

0	2	2	1	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1
2	0	1	1	2	2	1	1	1	2	1	1	1	2	1	1	2	1	2	1
2	1	0	1	2	2	1	1	2	1	1	2	2	2	1	2	2	1	2	1
1	1	1	0	1	1	1	2	2	1	1	1	1	2	1	1	1	1	1	1
1	2	2	1	0	1	1	1	1	1	2	2	1	1	2	2	1	1	1	1
2	2	2	1	1	0	1	2	2	1	2	1	1	2	1	1	1	1	2	2
1	1	1	1	1	1	0	1	1	1	1	1	2	1	1	1	1	1	1	1
2	1	1	2	1	2	1	0	1	1	1	1	1	1	1	2	1	1	1	1
1	1	2	2	1	2	1	1	0	2	1	2	2	1	1	1	2	1	1	1
1	2	1	1	1	1	1	1	2	0	1	1	1	1	2	2	1	2	1	1
1	1	1	1	2	2	1	1	1	1	0	1	2	1	1	2	2	1	2	2
1	1	2	1	2	1	1	1	2	1	1	0	1	1	1	1	1	1	2	1
1	1	2	1	1	1	2	1	2	1	2	1	0	1	1	1	1	1	1	1
1	2	2	2	1	2	1	1	1	1	1	1	1	0	1	1	1	2	2	1
1	1	1	1	2	1	1	1	1	2	1	1	1	1	0	1	1	1	1	2
1	1	2	1	2	1	1	2	1	2	2	1	1	1	1	0	1	1	1	2
1	2	2	1	1	1	1	1	2	1	2	1	1	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1	1	2	1	1	1	2	1	1	1	0	1	1
1	2	2	1	1	2	1	1	1	1	2	2	1	2	1	1	1	1	0	1
1	1	1	1	1	2	1	1	1	1	2	1	1	1	2	2	1	1	1	0

We see that all nodes are very close to each other. Every node can be reached in less than two steps. That makes it almost impossible to divide the variables into two groups such that the variables in the first group are closer to their groupmates than to the members of the other group. Doing the procedure described above in the section where the third crossover strategy is explained is just a very expensive way to get a more or less random map. Basically, it is doing the same as strategy 2. That is also reflected by the results of our experiments. It would be interesting to see, whether this strategy has advantages when we apply our algorithm to simpler problems.

- Does crossover method 3 gets better when we try different convergence rates?

0.1	0.2	0.3	0.4	0.5	0.6	convergence-rate
50	58	42	43	52	44	median

No, we can't identify any improvement. This confirms that our explanation on the previous item is along the right tracks.

How does our genetic algorithm compare to other algorithms that are known to be good for solving hard 3-SAT problems? Unfortunately, when we used Davis-Putnam to sort out problems that are not satisfiable, we saw right away that our approach is much much slower than the Davis-Putnam procedure (Selman 1996) or GSAT (Selman 1992), which has a comparable performance.

5 Used software and hardware

To enable us to implement all those various variations of genetic operators we implemented our own genetic algorithm toolbox in C. Furthermore, we implemented random K-SAT and the Davis-Putnam procedure (Selman 1996) to produce hard satisfiable 3-SAT problems. Also we implemented GSAT (Selman 1992) to have another randomized algorithm for comparison. The programs were running on the Sun-Workstations in the UNIX-cluster of Stanford University.

6 Conclusion

We showed that 3-SAT problems can be solved using genetic algorithms. But since the search space appears to be more or less unstructured, our approach has a hard time competing with established methods. Furthermore, we explored possible improvements of genetic operators. Here we have seen that a more flexible crossover is beneficial for the 3-SAT problem. Other ideas were shown to be irrelevant when solving hard 3-SAT problems.

7 Future Work

In the future it would be interesting to see how our algorithm scales. We would try to solve 3-SAT problems that have many more variables than the problems used here. This would necessarily involve running many more generations and also bigger generations. Maybe for bigger problems our algorithm can compete with the established methods.

8 Acknowledgements

Finally I would like to thank Todd Han for many valuable discussions and comments and also Professor Koza for helpful discussions when I was searching for an interesting and at the same time feasible project.

References

[Selman 1996] Selman, B., Levesque, H.J., Mitchell, D. 1996. Generating hard satisfiability problems. *Artificial Intelligence 81.* Pages 17-29.

[Cook 1971] Cook, S. A. 1971. The complexity of theorom-proving procedures. *Proceedings 3rd Annual ACM Symposium on the Theory of Computing.* New York. Pages 151-158.

[Selman 1992] Selman, B., Levesque, H.J., Mitchell, D. 1992. A New Method for Solving Hard Satisfiability Problems. *Proceedings 10th National Conference on Artificial Intelligence.* Pages 440-446.

[Goldberg 1989] Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA. Addison-Wesley.

[Koza 1992] Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA. The MIT Press.