GGA: A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms

Kevin Tierney

Brown University, Department of Computer Science,
P.O. Box 1910, Providence, RI, 02912, USA
ktierney@cs.brown.edu

Supervised by Prof. Meinolf Sellmann

Abstract. Tuning parameters is a problem that is inherent to the development and efficient use of solvers. We propose a robust, inherently parallel genetic algorithm for the problem of configuring solvers automatically. In order to cope with the high costs of evaluating the fitness of individuals, we introduce a gender separation whereby we apply different selection pressure on both genders. Experimental results on a selection of SAT solvers show significant performance and robustness gains over the current state-of-the-art in automatic algorithm configuration.

1 Introduction

Practically all solvers have parameters effecting performance that are either fixed by the programmer or set by the user. In recent years, systems have been devised which automate the task of tuning parameters for a given set of training instances that are assumed to represent typical instances for the target algorithm.

There are several motivations for automatic parameter tuning, the first being that it is extremely time consuming to tune parameters. Leaving the configuration of solvers to a computer rather than doing it by hand not only saves time, it could lead to better results.

Moreover, it is conceivable that the existence of an effective tuning environment will cause algorithm developers to parameterize more aspects of their algorithms and thus leave more freedom for algorithmic solutions that are automatically tailored to the problems of individual users. In particular, many of the SAT solvers that are available today have parameters which cannot be set through the command line. These parameters have been fixed to values that the developers have found beneficial without knowledge about the particular instances for which a user may want to use the solver. Automatic parameter tuning allows solvers to adapt to the final environment in which they need to perform. After being shipped, rather than relying on default parameters, an algorithm can be tuned automatically for the common tasks it is actually used for, and without requiring the user to learn about the algorithm parameters. For this very reason, Cplex 11 now comes with an automatic performance tuning tool.

Another argument for automatic solver configuration regards our own science: when we reimplement algorithms to conduct experimental comparisons with competing approaches, it is not unreasonable to assume that scientists spend much more time tuning their own algorithm than the algorithms of their competitors. A fair comparison could be achieved if all algorithms were tuned by an independent system. We could thereby come closer to understanding the true potential of algorithmic approaches rather than the ability of solver development teams to tune their solvers well.

1.1 Existing Approaches

Several approaches exist in the literature for the automatic tuning of algorithms prior to GGA[1]. The first methods were created for tuning specific algorithms for a certain task. [14] devised a modular algorithm for solving constraint satisfaction problems (CSPs) and used a combination of exhaustive enumeration of all possible configurations and a parallel hill-climbing technique to automatically configure the system for a given CSP with an associated set of training instances. [5] classified local search (LS) approaches for SAT by means of context-free grammars and devised a genetic programming approach to select a good LS algorithm for a given set of SAT problems. [16] embedded a sequential parameter optimization approach in a wider framework for the design of evolutionary algorithms.

To tune the continuous parameters of general algorithms, [4] suggested an approach that determines good parameters for individual training instances. These parameters are found by trying configurations where parameters are at their extreme values and then fitting a regression function to the parameter/value tuples obtained in this way. The minimization of the resulting function yields a set of parameters for the given instance. A parameter set for the entire collection of instances was then obtained by averaging the parameter tuples for the individual instances.

Tuning problems with small sets of parameter configurations were considered in [3], a setting which is closely related to that in algorithm portfolios [7, 8]. In this case, it is possible to race the different algorithms against each other, whereby a statistical test is used to eliminate inferior algorithms before the remaining algorithms are run on the next training instance.

In [15], Oltean used evolutionary algorithms by means of linear genetic programming. The genome of an individual is an encoding of an actual C-program for the problem to be solved, and crossover and mutation operators are problem dependent. The linear genetic program generates new individuals which replace the current worst individual in the population.

The CALIBRA system, proposed by [2], starts with a factorial design of the parameters. Once these initial parameter sets have been run and evaluated, an intensifying local search routine is started from a promising design, whereby the range of the parameters is limited according to the results of the initial factorial design experiments.

The only system we know of that can configure arbitrary algorithms with very large or even infinite numbers of possible configurations was proposed by [10]. Their system, called ParamILS, conducts an iterated local search, whereby a special technique is used to limit the number of training instances that need to be run for each parameter set by focusing the test runs on promising parameter sets. In particular, a new set of parameters is not considered better than the current best until it has been evaluated on at least as many training instances as the current best. If a very large set of training instances is available, this approach allows quick movement through the search space while still avoiding an “over-tuning” effect which would be caused by considering few training instances only.

1.2 Our Approach

CALIBRA and ParamILS have shown that automatic configuration of algorithms is possible, and can, in fact, lead to massive improvements over hand-tuned parameter sets. Based on these successes, we aim to provide a configuration system which is very robust and provides high-quality parameter sets in an affordable amount of time, potentially by exploiting parallelism which is becoming more and more widely available given the current trends in hardware technology.

To this end, we propose a genetic algorithm for the problem of configuring solvers. There are two main reasons for this choice of approach. First of all, genetic algorithms are known to be very robust with respect to optimization problems that have undesirable objective landscapes [6]. Note that, in ordinary optimization, we usually have the freedom to adjust the objective in such a way that it is better suited for sequential local search which often yields good solutions faster than population-based approaches. In contrast, in our setting, where the target algorithm is given and the effect of changing parameters is a priori unknown, we must be able to cope with whatever objective landscape we encounter. The other reason is that genetic algorithms are inherently parallel. When trying to assess which individuals are competitive (the most time-intensive step in solver configuration), genetic algorithms allow us to race them against each other. Therefore, the time spent for the evaluation is determined by the good parameter sets, and this saves a lot of time in practice. In order to really exploit this last aspect, we introduce the concept of gender in the genetic algorithm. Before we apply this idea to solver configuration, we will explain the potential benefits of gender separation in genetic algorithms in the following section.

2 A Gender-Based Genetic Algorithm

The idea to exploit genders in genetic algorithms has been considered before in various publications, inspired by nature's example.

2.1 Related Work

In [11], Lis and Eiben use multiple genders for multi-objective optimization. Each gender is associated with one of the multiple objectives, and the fitness of each individual is evaluated according to the gender-specific fitness function. Cross-over is limited to mating individuals of different gender. This latter restriction used for single-objective problems is introduced in [17] which uses a gendered genetic algorithm to solve graph partitioning problems.

[13] discusses that, in nature, mate choice is more likely to guide evolution than natural selection. [19] introduces two genders where individuals of the first are evaluated according to standard fitness, while individuals of the second gender are associated with a second criterion, a so-called cooperative fitness. This second criterion depends on the potential mating partner. Mating couples are formed by selecting the fittest individuals from the first gender and mating them with good cooperative partners from the second.

Finally, Vrajitoru experiments with a population that consists of different gender types (self-fertilizing, sexual, and hermaphrodite) [20] .

2.2 Gender-Specific Selection Pressure

Previously, gender separation has been realized only as a restriction of the way in which mating pairs are formed. We propose to take the gender separation beyond just the formation of parent couples. Instead, we propose to apply *different selection pressure* on the two gender populations. In particular, we apply intra-specific competition *only* in one part of the population. Individuals in this group must compete for the right of mating, and only the fittest in each generation win the right to mate with some of the individuals of the opposite gender. The individuals in this other group are not subjected to intra-specific selection.

With our application of tuning solver parameters in mind, we choose this setting for two reasons. First, the runtime of our algorithm will largely depend on the time that it takes to evaluate the fitness of individuals. Note that this requires running the given solver on some benchmark instances. By selecting only a small portion of the fittest individuals in one subgroup of the population, we can save a substantial number of fitness evaluations compared to the traditional way of applying a fitness ranking on all individuals.

Second, because of the expected very large cost to evaluate the fitness of an individual, in our application we will not be able to sustain a very large number of individuals in the population. Consequently, we cannot afford to lose partially good genes just because they first occurred in a less

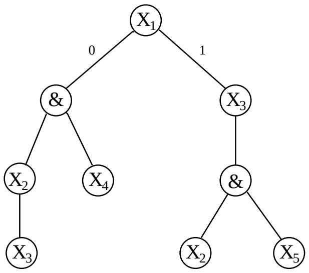

Fig. 1. Variable Tree.

fit individual. By randomly assigning new offspring to one of the two groups, we increase the chance that good genes survive as they may occur in individuals which belong to the non-competitive part of the population. That is, the individuals in this group serve as a "variety store" of potentially beneficial gene-collections which are only indirectly subjected to the selection process. Note that recombining information from non-competitive solutions has been found beneficial in other contexts as well – for example when solving vehicle routing problems [18].

The individuals in the non-competitive part of the population can of course still be expected to improve in average fitness over time. However, this happens without the need to evaluate directly the fitness of these individuals: Less fit parents are likely to have less fit children and half of those can be expected to belong to the competitive part of the population. These children, if they are indeed not fit enough, will not be able to propagate their genes into the grandchild-generation. Consequently, the chances that the genetic line of unfit parents will survive are greatly diminished. Note that this dynamic system agrees more closely with our modern understanding of evolution where the "survival of the fittest" is viewed more as the struggle for survival of genetic lines rather than that of individual beings.

2.3 Variable Trees

Apart from the fact that the introduction of gender will help reduce the number of fitness evaluations, in nature the invention of gender went hand-in-hand with the invention of recombination. Only sexual organisms have separate chromosomes which can be combined in arbitrary ways. This recombination process is realized by individuals having multiple sets of the entire genome – in humans for example there is one set of chromosomes from each parent. In our design we did not go quite as far as to introduce multiple copies of the genome. However, the idea that certain genes ought to be inherited as a set while others may be permuted and recombined arbitrarily aligns nicely with our expectation that certain groups of parameters will be linked more closely than others.

Therefore, we allow the user to define the design of the genome by passing a specific structure of the problem variables. With our application in mind, the variables will of course be the parameters of the solver to be tuned. In particular, to couple and de-couple variables (parameters), the user passes a variable structure which is inspired by And/Or-trees (see, e.g., [12]). The idea is that And-nodes separate variables that can be optimized independently.

To specify the parameters of a solver and their relation, we distinguish three types of variables: continuous and integer variables, both associated with an upper and a lower bound and categorical variables that come with an explicitly given list of feasible values. A *variable tree* is a tree where:

- Each node is labelled with a variable or the additional label "&", and each problem variable is associated with at least one node.

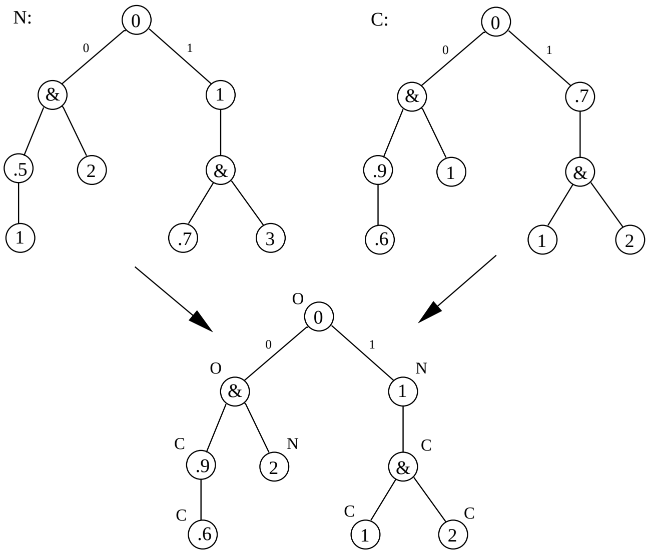

Fig. 2. Crossover Operator.

- Nodes associated with continuous or integer variables have at most one child, and And-nodes have at least two child-nodes.
- The children of categorical nodes partition the set of values that their parent variable can take. Branches leading to the children are labelled by the respective value(s) of the categorical variable.

In Figure 1, we illustrate a variable tree for the minimization of the function $x_1\left(\frac{x_2\sin(\pi(x_2-x_3))}{x_3}+(x_4-2)^2\right)+2(1-x_1)\left(|\frac{x_5}{x_3}-7|+(x_2x_3-1)^2\right)$ whereby x_1 is categorical and takes values 0 or 1, x_2, x_3 are continuous and take values in $[\frac{1}{2}, 1]$, and x_4, x_5 are integer variables taking values in $\{1, 2, 3\}$. This function is used merely to show an interesting semantic tree for illustration purposes. We see that the structure reflects that, once x_1 is set to 0, the pair of variables x_2, x_3 can be optimized independently of x_4. Also, once x_1 is set to 1 and x_3 has been assigned a value, variables x_2 and x_5 can be optimized independently.

2.4 A Gender-based Genetic Algorithm

We now have all concepts in place to describe our Gender-based Genetic Algorithm (GGA) for the automatic configuration of solvers. GGA uses parameters (X, P, M, A, S) which are used in the following way:

- **Initialization:** First, we randomly initialize the population and assign a gender C (for competitive) or N (for non-competitive) and an “age” of 1 to A years uniformly at random to each individual. In our experiments, we set A to 3.
- **Mating Rules:** Among the individuals with gender C, we select the top X% (in our experiments we set X to 10%). These have gained the right to mate in this season. $200/A$% of individuals of gender N are assigned uniformly at random to one of the mating individuals of gender C. The individuals of gender C then mate with all individuals of gender N which have been assigned to them.
- **Crossover:** Each mating of a couple results in one new individual with age 0 and random gender. The genome of the offspring is determined by traversing the variable tree top-down (compare

```
Crossover(ParameterTrees $T_C, T_N, T_3$)

curNode ← rootOf($T_3$), nodeC ← rootOf($T_C$), nodeN ← rootOf($T_N$)
if (type[curNode]=And) OR (value[nodeC]=value[nodeN]) then
    label[curNode] ← O, value[curNode] ← value[nodeC]
else
    if rand() mod 2 = 0 then
        label[curNode] ← C, value[curNode] ← value[nodeC]
    else
        label[curNode] ← N, value[curNode] ← value[nodeN]
    end if
end if
S ← $\{curNode\}$
while (S $\neq \emptyset$) do
    curNode ← $pick(S)$, S ← S $\setminus \{curNode\}$
    for all childNodes of curNode do
        S ← S $\cup \{childNode\}$
        nodeC ← correspondingNode($T_C$, childNode)
        nodeN ← correspondingNode($T_N$, childNode)
        if label[curNode]=O then
            if type[childNode]=OR then
                if value[nodeC]=value[nodeN] then
                    label[childNode] ← O, value[childNode] ← value[nodeC]
                else
                    if rand() mod 2 = 0 then
                        label[childNode] ← C, value[childNode] ← value[nodeC]
                    else
                        label[childNode] ← N, value[childNode] ← value[nodeN]
                    end if
                end if
            else
                label[childNode] ← O, value[childNode] ← value[nodeC]
            end if
            continue
        end if
        if label[curNode]=C then
            parLabl ← C, parVal ← value[nodeC]
            oParLabl ← N, oParVal ← value[nodeN]
        else
            parLabl ← N, parVal ← value[nodeN]
            oParLabl ← C, oParVal ← value[nodeC]
        end if
        if rand() mod 100 $< \delta$ then
            label[childNode] ← oParLabl, value[childNode] ← oParVal
        else
            label[childNode] ← parLabl, value[childNode] ← parVal
        end if
    end for
end while
```

Algorithm 1: Crossover Algorithm.

with Algorithm 1 and Figure 2). A node can be labelled O ("open"), C, or N. If the root is an And-node, or if both parents agree on the value of the root-variable, we label it O. Otherwise, we randomly assign it label C or N (Lines 5 and 23). The algorithm continues by looking at the children of the root (and so on for each new node). If the label of the parent node is C (or N) then with probability P% we also label the child with C (N), otherwise with N (C) (Line 41). In our experiments we set P to 90%.

  Finally, the variable assignment associated with the offspring is given by the values from the C (N) parent for all nodes labelled C (N). For variable-nodes labelled O both parents agree on its value, and we assign this value to the variable. Note that this procedure effectively combines a uniform crossover for child-variables of open And-nodes in the variable tree (thus exploiting the independence of different parts of the genome) and a randomized multiple-point crossover for variables that are more tightly connected.

- **Mutation:** As a final step to determine the offspring's genome, with probability M% we mutate the value of each variable (in our experiments, we set M to 10%). If we mutate a categorical variable, we choose a new value in its domain uniformly at random. For continuous and integer variables, we choose a new value according to a Gaussian distribution where the current value marks the expected value and the variance is set as S% of the variable's domain. In our experiments, we set S to 5%.
- **Death:** After the new offspring is created, all individuals' ages are increased by 1. Those with age greater than A are removed from the population. In combination with the mating rules that only $200/A$% of individuals of gender N mate in every season, this stabilizes the total population size.

Before use this algorithm for the configuration of solvers, we first test it on some generic optimization functions of three different types: $f_1 = \sum_i (x_{2i}x_{2i+1} - p_i)^2$,
$f_2 = \sum_i (x_{3i}x_{3i+1}x_{3i+2} - q_i)^2$, and $f_3 = \sum_i ((x_{3i}x_{3i+1} - v_i)^2 + (x_{3i}x_{3i+2} - w_i)^2)$, whereby all variables take integer values in $\{0, \ldots, 15\}$ and constants $p_i, v_i, w_i \in [0, 15^2]$, $q_i \in [0, 15^3]$ are chosen uniformly at random.

For each of these functions we compare three different variable trees that determine the genome structure: The completely independent structure (IND) with one And-node at the root and all variables as its children, the completely dependent structure (DEP) where all variables form one long chain, and the semantic structure (SEM) that results from the static analysis of each of the functions (with independent tuples (x_{2i}, x_{2i+1}) in f_1, $(x_{3i}, x_{3i+1}, x_{3i+2})$ in f_2 and f_3, whereby in the latter x_{3i+1} and x_{3i+2} are independent once x_{3i} has a value). Table 1 compares Gender-based GA (GGA) with the GA library GALib from [21]. The genome for the latter is determined by representing each variable as a four-bit string which tests showed to yield the best results. The cross-over and mutation probabilities were set to 1 and 0, respectively, which our own parameter tuner confirmed to be the best parameter settings for this optimization system. For our algorithm, we chose the parameter the set $(10, 90, 10, 3, 10)$. That is, only the top 10% of competitive individuals are allowed to mate in each season; with probability 90% a child-variable inherits the value from the same parent as its parent-variable in the variable tree; the mutation-rate is 10%; individuals die after 3 mating seasons; and nodes in the child variable trees will randomly change their assignment from their parent's with probability 10%.

The results clearly show the benefits of introducing gender to reduce the number of function evaluations. Only half of the population is evaluated in each season. Moreover, we see that applying different selection pressure on both genders results in significantly improved solution quality. For example, when optimizing f_3 with a population size of 500 over 100 generations, the gender-based approach using the semantic genome structure (SEM) on average gives solution values around 1,250

	GGA			
Prob-Pop-Gen	IND	DEP	SEM	GA
f_1-1000-25	1.8(13.7)	29.8(13.7)	**1.44**(13.8)	7.19(26.0)
f_1-500-50	0.59(13.4)	28.9(12.8)	**0.4**(13.3)	4.56(25.5)
f_1-2000-25	1.54(27.1)	29.3(27.7)	**1.29**(27.6)	6.67(52.0)
f_1-1000-50	0.46(26.0)	28.7(26.1)	**0.31**(26.0)	4.41(51.0)
f_1-500-100	0.16(25.1)	28.3(25.0)	**0.13**(25.5)	2.69(50.5)
f_2-1000-25	1442(13.7)	6075(13.8)	**1229**(13.4)	4962(26.0)
f_2-500-50	392(13.3)	5273(13.0)	**340**(13.0)	3127(25.5)
f_2-2000-25	**1104**(27.4)	5121(25.3)	1119(27.6)	4405(52.0)
f_2-1000-50	307(25.6)	5886(27.5)	**304**(25.8)	3184(51.0)
f_2-500-100	141(25.3)	4830(25.4)	**124**(26.0)	2002(50.5)
f_3-1000-25	**16.2**(13.6)	18.5(13.9)	16.4(13.7)	43.2(26.0)
f_3-500-50	**5.75**(13.2)	8.01(13.5)	6.33(13.7)	31.9(25.5)
f_3-2000-25	14.7(27.5)	15.9(27.5)	**13.7**(27.2)	41.3(52.0)
f_3-1000-50	**4.86**(25.9)	6.40(26.5)	5.27(25.8)	31.8(51.0)
f_3-500-100	1.33(26.5)	1.7(27.1)	**1.25**(25.8)	22.5(50.5)

Table 1. Numerical Results for generic function minimization. We give the average solution value and, in parenthesis, the number of function evaluations (both in thousands) for 50 runs.

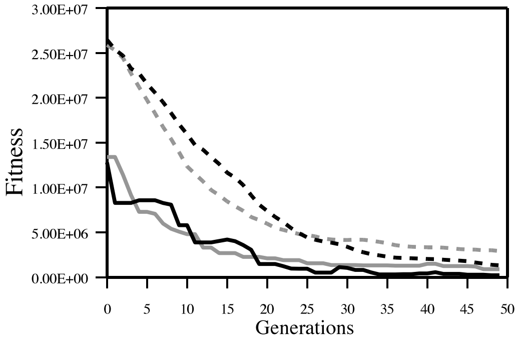

Fig. 3. Average (dashed) and best (solid) fitness for the competitive (black) and non-competitive (gray) populations when optimizing f_2 (averages over 25 runs).

at the cost of 25,800 fitness evaluations. The standard genetic algorithm performs 50,500 evaluations and still only achieves average solution values of 22,500.

The effectiveness of our variable tree is also confirmed. The semantic tree resulted in better performance in nearly all of our tests, and was only slightly outperformed by IND in the few instances where the latter performed better.

In Figure 3 we show how the average and the best fitness in both parts of the population evolve over the course of several generations. Even though it may not have been expected, the average and best fitness in the non-competitive part of the population are not worse than in the competitive part. This is of course caused by the fact that gender is assigned randomly and the fitness of new individuals (no matter to which gender they belong) is determined by the fitness of both parents. Consequently, we found that the overall best solution was equally often found in both genders.

3 Automatic Solver Configuration

By representing the genome as a variable tree which is defined by the user of the configuration system, the outlined genetic algorithm can be applied directly for the task of tuning solver parameters. Note that our approach allows a separation of setting the structure of the parameters (which would typically be done by the developer of the solver) and selecting the training instances (which are

	GA		GGA		Improvement
	Avg.	Std. Dev	Avg.	Std. Dev.	[%]
Time	23.5K	4.48K	926	1.12K	96
Quality	1.79	1.57	0.07	0.01	96

Table 2. Experimental results of a standard GA vs. GGA for configuring SAPS with a cutoff of 10 seconds. We report the average total CPU time over 10 runs of the tuners as well as the average solution quality. All times are given in seconds.

best selected by the user of the solver). While using the same parameter categories as ParamILS, the variable tree representation does not require discretization of continuous parameters and offers a richer way of specifying parameter correlations.

To apply our genetic approach for solver configuration we only need to specify the function which selects the top X% of the competitive part of the population. We could of course run all individuals (i.e., parameter sets) on all training instances and then select the best ones. However, this costs a lot of time, especially when the task is to minimize the expected average runtime of the target algorithm, which is the most common evaluation criterion for solvers. In this case, we run the different parameter sets in parallel, and as soon as the best X% have finished, we interrupt all remaining runs. This way, the time spent is largely determined by the quality of the very good parameter sets.

For larger population sizes, creating too many parallel threads or processes is not appealing. In these instances, we partition the competitive population into smaller groups (typically between 5 and 10 members per available CPU) and select the best X% of this subgroup by running them in parallel. This is repeated until all groups have found their winners. In our experiments, we used groups of 8.

The size of the training set is also an important consideration. Many training instances are desirable so as to avoid an over-tuning effect where we find parameter sets which work particularly well for the training instance but do not generalize. On the other hand, evaluating each parameter set on all available instances costs a lot of time. [10] proposed a focused approach which results in more evaluations of better parameter sets. In our setting, we could employ the racing framework from [2] for selecting the top competitive individuals. However, we take the practical standpoint that we will normally not have access to as many training instances as we would like (which is necessary to obtain the theoretical guarantees that the probability of over-tuning converges to zero). Therefore, in each mating season we select a random subset of all available training instances and race the competitive individuals on those. In subsequent generations we also increase the relative size of the subset linearly, so that the comparison of different parameter sets becomes more and more accurate the better the parameter sets become.

4 Numerical Results

4.1 Gender Separation for Solver Configuration

We first compare our configurator with a “standard” genetic algorithm to demonstrate the advantages of the gender separation for configuration. We use a standard genetic algorithm along with our variable tree and crossover operations.

We compute 40 generations with a population of 30 individuals for both genetic algorithms. The standard GA does not perform a gender separation and uses a different mating scheme: Each individual in the population is evaluated in each generation. The oldest third of the population is replaced by new offspring. This is formed by mating two individuals from the population. The probability of an individual being chosen for mating is proportional to its fitness.

In Table 2 we report the configuration time as well as the runtime of the target solver when using the final set of parameters for both configurators. The training was done on a set of 113 SAT instances, the quality of the configuration was evaluated on a set of 100 different SAT instances (see [10]). The tests were run on a AMD Athlon 64 3000+ CPU with 2 GB RAM.

We see that GGA works 20 times faster and returns configurations which are much better. Compared to running SAPS with the parameters found by the GA, SAPS requires only about 4% of the runtime using the parameter set returned by GGA. The great reduction in configuration time is mainly due to the fact that it is sufficient for GGA to find the top 10% of the competitive individuals. This can be done by racing parameter sets against each other, rather than evaluating them all. Since bad parameter settings can take a lot of time (the effect is softened by the cutoff of 10 seconds, but still significant), GA wastes a lot of time evaluating bad solver configurations.

This alone does not explain the greatly improved quality, though. As seen earlier in our preliminary experiments optimizing a generic function, the gender separation improves the solution quality by providing a store of genes which diversify the search and prevent us from getting stuck in local optima even though we aggressively select only the top 10% of the competitive individuals for mating.

4.2 GGA vs. ParamILS

We evaluate our system on two of the same target algorithms that [10] used to show that their ParamILS approach outperforms the CALIBRA system from [2]. These are the previously considered SAPS solver and SAT4J, a systematic solver for SAT. The benchmark set for SAPS consists of 113 SAT instances for training and 100 different instances for testing. Since SAPS is a randomized solver, the instances are paired with 10 different seeds each. We used the SWGCP benchmark set from [10] for configuring SAT4J. It consists of 1000 SAT instances for training and 1000 different instances for testing. In addition, we compared our solver with ParamILS on the target algorithm SPEAR, another systematic SAT solver. SPEAR was tuned by ParamILS in [9]. Again, we used the SWGCP benchmark for training and testing. The parameter tree structure of these solvers is given in the Appendix in figures 4, 5, 6 and 7.

Algorithms SAPS and SPEAR were run on a 32bit Linux Intel Core2 Quad CPU Q6600 with 2.4GHz and 3GB RAM, SAT4J was run on a 64bit Linux Intel Xeon with 2.8GHz and 8GB RAM. The objective in all experiments is to minimize the mean runtime of the solver. We ran GGA for 100 generations with a population size of 200 for SAPS and 50 for SPEAR. For SAT4J, we ran GGA for 70 generations with a population of size 60. This resulted in total CPU times for configuration that never exceed the 10h cutoff which was used for ParamILS when configuring SAPS and SPEAR, and a 20h cutoff for SAT4J.

In Table 3 we show the results of 20 configuration runs of ParamILS[1] and our Gender-based genetic algorithm (GGA) when configuring solvers SAPS and SAT4J. We distinguish two different versions of SAT4J: SAT4J with a full parameter set some of which allow the solver to return unsatisfying SAT solutions, and SAT4J* with a limited parameter set all of which force the solver to return only feasible solutions.

The table gives the final average computation time for the instances in the training as well as the test set. Comparing the training and test performances, we see that GGA very accurately assesses the expected performance. For SAPS and SAT4J*, ParamILS also assesses its performance quite accurately. However, for SAT4J, ParamILS returns parameters which perform quite badly on the training set and perform a lot better on the test set. For example, in configuration run 1 ParamILS

[1] Thanks to Frank Hutter for providing support of ParamILS and the various solvers!

	ParamILS		GGA	
Run	Train	Test	Train	Test
1	51.55	52.12	37.25	35.44
2	53.91	51.45	46.15	41.69
3	55.31	50	55.85	49.52
4	55.11	51.8	36.07	33.4
5	53.72	50.91	46.15	40.1
6	54.96	52.4	35.68	33.56
7	54.67	52.79	34.69	32.2
8	55.11	49.91	37.54	32.76
9	56.24	51.09	38.24	35.42
10	56.26	51.31	34.7	33.53
11	55.02	52.26	35.99	34.52
12	54.29	51.61	36.1	33.99
13	54.31	51.42	36.35	33.59
14	56.58	52.31	37.42	34.58
15	57.38	54.15	38.79	36.66
16	57	54.09	52.98	52.25
17	56.31	52.6	35.78	32.68
18	58	53.73	38.59	35.06
19	54.52	51.83	35.83	33.9
20	61.47	55.85	36.57	34.56
∅	55.6	52.2	39.3	36.5
σ	2.0	1.44	6.0	5.5

	ParamILS		GGA	
Run	Train	Test	Train	Test
1	3.65	0.99	1.07	1.07
2	1.06	1.07	1.05	1.06
3	1.07	1.07	1.06	1.06
4	0.99	1.05	1.06	1.07
5	5.11	2.22	1.07	1.14
6	1.04	1.04	3.20	3.90
7	5.29	2.31	1.05	1.05
8	6.27	2.38	1.06	1.06
9	1.05	0.53	1.04	1.05
10	1.06	1.07	1.05	1.05
11	1.17	1.06	1.14	1.06
12	1.04	1.05	1.05	1.13
13	1.05	1.06	1.07	1.07
14	1.05	1.05	1.05	1.06
15	1.04	1.05	1.05	1.06
16	6.34	6.83	1.08	1.08
17	5.17	5.35	1.07	1.07
18	4.70	4.32	1.05	1.06
19	5.57	5.20	1.06	1.06
20	5.03	4.97	1.05	1.06
∅	2.94	2.28	1.17	1.21
σ	2.2	1.92	0.48	0.63

	ParamILS		GGA	
Run	Train	Test	Train	Test
1	2.20	2.86	3.19	2.92
2	5.06	5.41	3.99	3.66
3	4.70	5.78	3.01	2.92
4	2.44	2.58	2.90	2.99
5	2.56	2.57	3.06	3.14
6	2.48	2.76	2.92	2.93
7	2.52	2.77	3.00	2.95
8	2.55	2.66	2.90	2.92
9	2.80	2.96	3.01	3.00
10	2.57	2.92	2.96	2.93
11	2.45	3.92	2.87	2.93
12	4.90	5.05	2.87	2.79
13	2.49	3.93	3.01	2.81
14	2.41	2.92	2.96	2.81
15	5.25	5.93	3.01	2.70
16	4.21	4.36	3.29	3.12
17	4.89	5.39	4.91	4.88
18	2.54	2.63	3.34	3.24
19	2.48	2.55	2.92	2.92
20	2.45	2.42	3.04	3.06
∅	3.2	3.62	3.16	3.08
σ	1.12	1.24	0.48	0.47

Table 3. Experimental Results for Configuring SAPS (left), SAT4J (middle), and SAT4J* (right).

	ParamILS		GGA		Welch's t-test		Improvement
Problem	Train	Test	Train	Test	t(1-tail)	t(2-tail)	[%]
SAPS [ms]	55.8 (2.3)	52.4 (1.78)	38.8 (5.5)	36.0 (5.0)	< 0.01	< 0.01	31.30
SPEAR [s]	1.58 (0.088)	1.49 (0.087)	1.58 (0.086)	1.50 (0.077)	0.33	0.65	-0.67
SAT4J [s]	2.85 (2.12)	2.38 (1.97)	1.25 (0.67)	1.29 (0.76)	0.01	0.01	45.80
SAT4J* [s]	3.32 (1.11)	3.74 (1.28)	3.28 (0.89)	3.20 (0.81)	0.04	0.08	14.4

Table 4. Experimental results of ParamILS and GGA for various solvers. We give mean solver times on the training and test sets and, in brackets, the standard deviation over 20 configuration runs. We used Welch's t-test with the null hypothesis "the test time results of ParamILS are not different than the test time results of GGA."

finds a parameter set which actually works very well on the test set, although the performance on the training set is quite miserable. In configuration run 7, ParamILS also finds a set of parameters which actually works better on the test set than on the training set. However, the test performance is almost 100% worse than the average performance achieved by GGA.

In terms of solution quality, GGA achieves significant improvements, outperforming ParamILS on all three configuration tasks. For SAPS, GGA's worst parameter set results in an average performance of 52.25 ms per test instance, which is still better than the average ParamILS parameter set which requires 52.4 ms. Comparing average test performances (see Table 4), GGA's parameters improve those of ParamILS when configuring SAPS by more than 30%. For SAT4J, GGA returns parameters which are on average 45% better than those provided by ParamILS, and for SAT4J* by over 14%. Considering that configuration is essentially a complex optimization problem, this is a big margin. We are not aware of any other application where a population-based approach outperforms a state-of-the-art sequential local search algorithm that significantly. For comparison, [10] reported a 19% improvement over CALIBRA when configuring SAPS so as to minimize median runtime.

Looking at the standard deviation over the different configuration runs, we see that the genetic approach performs much more robustly than iterated local search whose standard deviation is at times as large as its mean. We attribute the solid performance of GGA to the robustness of genetic algorithms which are known to cope well even when objective landscapes are noisy and rugged as can be expected in algorithm configuration. Note that ParamILS was reported to find parameter sets which already improve on the manually set SAPS defaults by three to four orders of magnitude. The great additional improvement by GGA is fairly surprising and shows just how hard it actually is to find near-optimal parameters.

Finally, in Table 4 we also report the results when configuring the SPEAR program. We find that both ParamILS and GGA perform equally well for this configuration problem. We have no actual lower bounds on the performance that can be achieved by SPEAR, but we conjecture that there is simply not much room for improving this algorithm any further.

5 Conclusion

We considered the problem of automatically configuring solvers. We proposed a genetic algorithm for this task and showed that it robustly provides high quality parameter sets which can significantly improve over those found by the pioneering system from [10]. To specify and exploit the dependencies of parameters, we introduced a special structure which results in an automatic de-coupling of independent parameters in the cross-over operator.

To improve performance, our approach introduced a gender separation which we believe to be of interest for genetic algorithms in general, especially when population sizes are small and the optimization costs are largely determined by the number of fitness evaluations. What makes a genetic approach appealing for our application is its robustness and inherent parallelism. Our preliminary parallelization already resulted in substantial speed-ups. We are currently working on an efficient parallelization of our code which will provide the practical basis for configuring solvers that require more computation time. As future work, we are considering to locally improve good parameter sets which have been found, thus transforming our genetic algorithm into a memetic algorithm.

References

1. Ansotegui, C., Sellmann, M., Tierney, K.: A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. Proceedings of the 15th intern. Conference on the Principles and Practice of Constraint Programming (CP), Springer LNCS 5732, pp. 142-157, 2009.
2. Adenso-Diaz, B., Laguna, M.: Fine-tuning of Algorithms using Fractional Experimental Design and Local Search. Operations Research, 54(1):99–114, 2006.
3. Birattari, M., Stuetzle, T., Paquete, L., Varrentrapp, K. A Racing Algorithm for Configuring Metaheuristics. GECCO, 11-18, 2002.
4. Coy, S.P., Golden, B.L., Runger, G.C., Wasil, E.A. Using Experimental Design to Find Effective Parameter Settings for Heuristics. Journal of Heuristics, 7(1):77–97, 2001.
5. Fukunaga, A. Automated discovery of local search heuristics for satisfiability testing. Evolutionary Computation, 16(1):31–61, 2008.
6. Goldberg, D.E. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, 1989.
7. Gomes, C., Selman, B. Algorithm Portfolios. Artificial Intelligence, 126(1–2):43–62, 2001.
8. Huberman, B., Lukose, R., Hogg, T. An Economics Approach to Hard Computational Problems. Science, 265:51–54, 2003.
9. Hutter, F., Babić, D., Hoos, H.H., Hu, A.J. Boosting Verification by Automatic Tuning of Decision Procedures. FMCAD, 27–34, 2007.
10. Hutter, F., Hoos, H.H., Stützle, T. Automatic Algorithm Configuration based on Local Search. AAAI, 1152–1157, 2007.
11. Lis, J.,Eiben, A.E. A Multi-Sexual Genetic Algorithm for Multiobjective Optimization. IEEE International Conference on Evolutionary Computation, 59–64, 1997.
12. Marinescu, R., Dechter, R. And/Or Branch-and-Bound for Graphical Models. IJCAI, 224–229, 2005.

13. Miller, G.F., Todd, P.M. The Role of Mate Choice in Biocomputation. Evolution and Biocomputation, 169–204, 1995.
14. Minton, S. Automatically Configuring Constraint Satisfaction Programs. Constraints, 1(1):1–40, 1996.
15. Oltean, M. Evolving evolutionary algorithms using linear genetic programming. Evolutionary Computation, 13(3):387–410,2005.
16. Preuss, M., Bartz-Beielstein, T. Sequential Parameter Optimization Applied to Self-adaptation for Binary-coded Evolutionary Algorithms. Parameter Setting in Evolutionary Algorithms: Studies in Computational Intelligence, 91–119, 2007.
17. Rejeb, J., AbuElhaij, M. New Gender Genetic Algorithm for Solving Graph Partitioning Problems. Curcuits and Systems, 1:444–446, 2000.
18. Rochat, Y.,Taillard, R.D. Probabilistic Diversification and Intensification in Local Search for Vehicle Routing. Journal of Heuristics, 1:147–167, 1995.
19. Sanchez-Velazco, J., Bullinaria, J.A. Gendered Selection Strategies in genetic Algorithms for Optimization. UKCI, 217–223, 2003.
20. Vrajitoru, D. Simulating Gender Separation with Genetic Algorithms. GECCO, 634-641, 2002.
21. Wall, M. GAlib: A C++ Library of Genetic Algorithm Components. MIT, `http://lancet.mit.edu/ga`, 1996.

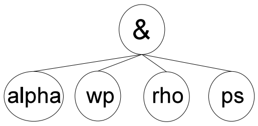

Fig. 4. Flat (independent) parameter tree for SAPS.

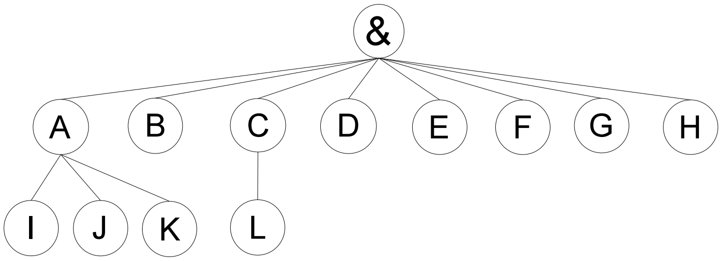

Fig. 5. Parameter tree for SAT4J.

6 Appendix

In the absence of specific information from the solver developers, we constructed our parameter trees based on descriptions of the parameters in each solver's documentation. These parameter trees are given in Figures 4, 5, 6 and 7. Each of these parameter trees has a root and node (shown by &) which does not take on any value. In the case of SAT4J, for example, using its documentation we determined that the parameter "period" (**L**) is only necessary when "order" (**C**) takes certain values and therefore has a dependence on "order" (**C**). It is possible that people with more intimate knowledge of the internal workings of the solvers could improve our variable trees to be more descriptive. The results of tuning these target algorithms are given in section 4. We refer to [10] for more information on the values the parameters can take. Note that in contrast to ParamILS, we do not discretize continuous parameters since they are supported by GGA.

Figure 5 displays the structure of the parameter tree for the solver SAT4J. The names of the parameters are listed below:

A. DSF
B. learning
C. order
D. SIMP
E. varDecay
F. claDecay
G. conflictBound
H. initConflictBound
I. activityPercent
J. maxLength
K. limit
L. period

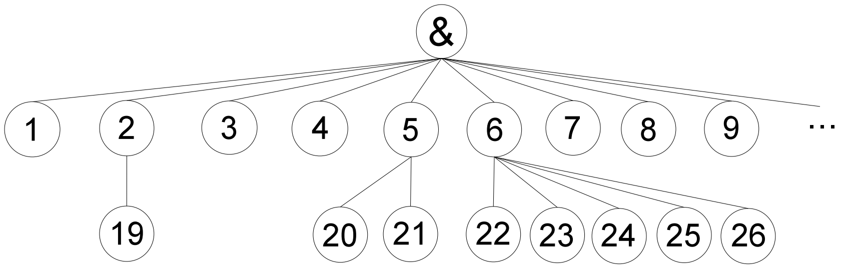

Fig. 6. Parameter tree for SPEAR (first half).

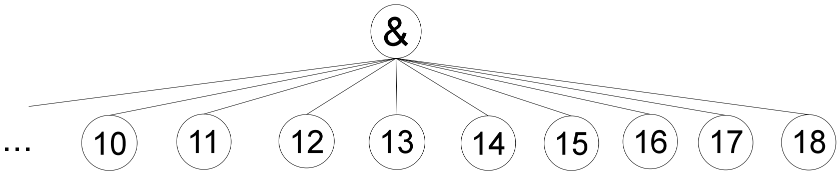

Fig. 7. Parameter tree for SPEAR (second half).

Figures 6 and 7 present the parameter tree for the solver SPEAR. The names of the parameters are listed below, and for more information on the function and domains of these parameters we refer to [9].

1. sp-var-dec-heur
2. sp-learned-clause-sort-heur
3. sp-orig-sort-heur
4. sp-clause-del-heur
5. sp-phase-dec-heur
6. sp-resolution
7. sp-variable-decay
8. sp-clause-decay
9. sp-restart-inc
10. sp-learned-size-factor
11. sp-learned-clauses-inc
12. sp-clause-activity-inc
13. sp-var-activity-inc
14. sp-rand-var-dec-freq
15. sp-rand-var-dec-scaling
16. sp-first-restart
17. sp-update-dec-queue
18. sp-use-pure-literal-rule
19. sp-clause-inversion
20. sp-rand-phase-dec-freq
21. sp-rand-phase-scaling
22. sp-res-order-heur
23. sp-max-res-lit-inc
24. sp-res-cutoff-cls
25. sp-res-cutoff-lits
26. sp-max-res-runs