Breeding Theorem Proving Heuristics with Genetic Algorithms

Simon Schäfer and Stephan Schulz

DHBW Stuttgart
mail@simon-schaefer.net, schulz@eprover.org

Abstract

First-order theorem provers have to search for proofs in an infinite space of possible derivations. Proof search heuristics play a vital role for the practical performance of these systems. In the current generation of saturation-based theorem provers like SPASS, E, Vampire or Prover 9, one of the most important decisions is the selection of the next clause to process with the given clause algorithms. Provers offer a wide variety of basic clause evaluation functions, which can often be parameterized and combined in many different ways. Finding good strategies is usually left to the users or developers, often backed by large-scale experimental evaluations. We describe a way to automatize this process using genetic algorithms, evaluating a population of different strategies on a test set, and applying mutation and crossover operators to good strategies to create the next generation. We describe the design and experimental set-up, and report on first promising results.

1 Introduction

First-order theorem provers have to search for proofs in an infinite search space. Modern calculi like superposition [1, 7] with strong redundancy criteria have reduced the branching factor in the search space, and efficient implementation techniques based e.g. on term- and clause indexing [15, 12, 13], together with the increase in hardware capabilities, allow us to penetrate much deeper into the space of possible derivations. However, this technical progress, while impressive, has less influence than the choice of good search control heuristics.

For modern saturation-based theorem provers like E [14, 11], SPASS [20] and Vampire [9, 5], heuristics have traditionally been developed and refined manually or semi-manually. This is based on the developers experience and, in more recent times, backed by a large number of empirical data, much of which is unfortunately rarely published. Due to the large number of possible parameter settings, it is very hard to explore the parameter space in full, and this has only been attempted for small sub-spaces (see e.g. [4]).

In this paper, we try to attack the problem in another way. Optimization in high-dimensional, complexly structured spaces is a problem not uniquely limited to automated theorem proving. In other domains, evolutionary approaches have been successful [3]. We adopt this approach, and apply genetic algorithms to the problem of generating good clause selection heuristics for the theorem prover E. E has a very general approach to clause selection, describing heuristics in a domain-specific language that allows the combination of an arbitrary number of different parameterized evaluation schemes. The evolving population of individuals consists of clause evaluation heuristics (i.e. words in the domain specific language, but abstractly represented as lists of nested syntactic elements). Each individual is evaluated on a small set of proof problems, and with a short search time limit. Fitness is assigned based on the number of solutions found. Promising individuals are then subject to mutation and cross-over, forming the next generation. This process is then iterated. At the end, the best heuristics are extracted, evaluated on larger problem sets and with more typical time limits, and compared to conventionally developed

strategies. This work is similar to Urban's BliStr [19], but differs in the use of genetic algorithms instead of simple hill-climbing. We also currently optimize only clause selection, while Urban applies his method to a number of additional parameters that guide e.g. preprocessing.

This paper is structured as follows: After this introduction, we discuss heuristics for ATP systems, and for E in particular. We also give a short overview on genetic algorithms. Then we describe the design and implementation of our system. The next chapter gives first results, before we conclude.

2 Background

2.1 Automated Theorem Proving and Proof Search

First-order theorem provers are controlled by a large range of search control options. We call a collection of settings for all such parameters a search control heuristic, or, interchangeably, a search strategy. For modern theorem provers, such a search control heuristic is composed of several different and, at least theoretically, independent components.

Saturating provers try to show the unsatisfiability of a set of clauses (and thus, via a proof by contradiction, the original hypothesis) by deriving the empty clause as an explicit witness of unsatisfiability. They are usually based on a variant of the given clause algorithm. In this algorithm, the clause set is split into a set P of processed and a set U of unprocessed clauses. Initially, all clauses are unprocessed. The prover moves one clause after the other from U to P, maintaining the invariant that all direct consequences between clauses in P have been computed (and added to U). In practice, contraction (i.e. the elimination or simplification of redundant clauses) also plays a critically important role, but this does not significantly affect our current discussion.

Search is controlled by the inferences allowed (or required) between any set of premises, and by the order in which clauses are selected for processing. The inferences are typically controlled by the term ordering, which is usually selected and remains static throughout one proof attempt, and the literal selection strategy, which is fixed for each clause when it is processed.

Clause selection in particular has been shown to have a significant influence on the performance of the prover. Most modern provers allow at least the interleaving of a best-first (lowest weight) and breadth-first (oldest clause) search, where the weight is usually based on symbol counting (smaller is better). E generalizes this concept, and allows the user to specify an arbitrary number of priority queues and a weighted round-robin scheme that determines how many clauses are picked from each queue. A major feature is the use of goal-directed evaluation functions. These give a lower weight to symbols that occur in the goal, and a higher weight to other symbols, thus preferring clauses which likely have a connection to the conjecture. As an alternative, E can also *learn* good clause evaluations from previous proof experience [10].

Thus, an individual clause selection strategy can be quite complex. Fig. 1 shows the textual rendering of one of the best clause selection heuristics used in E. This particular example sets up 5 priority queues, 3 based on goal-directed strategies, one based on first-in/first-out, and one preferring small goal-derived clauses. Looking further into the first weight function, the first parameter selects goal-derived clauses over all others, the second is the relative factor applied to symbols from the conjecture, the next for represent the base values of non-constant function symbols, constants, predicate symbols, and variables, in that order. The last three are factors applied to the weights of maximal (in the term ordering used to control superposition) terms, maximal literals, and positive literals.

```
(1*ConjectureRelativeSymbolWeight(SimulateSOS,
        0.5, 100, 100, 100, 100, 1.5, 1.5, 1),
 4*ConjectureRelativeSymbolWeight(ConstPrio,
        0.1, 100, 100, 100, 100,1.5, 1.5, 1.5),
 1*FIFOWeight(PreferProcessed),
 1*ConjectureRelativeSymbolWeight(PreferNonGoals,
            0.5, 100, 100, 100,100, 1.5, 1.5, 1),
 4*Refinedweight(SimulateSOS,3,2,2,1.5,2))
```

Figure 1: Example clause evaluation heuristic

This would be one example of an individual in our evolving population. Genetic operators allow modification of the numerical parameters, of the contributing individual evaluation functions, and of the symbolic (finite domain) parameters.

2.2 Genetic Algorithms

Genetic algorithms address optimization problems by evolving a population of potential solutions (or *individuals*) over several generations. In each generation, individuals are evaluated, resulting in a *fitness* score. A *selection* step preferably selects fitter individuals for contributing to the next generation. The next generation is created from the previous one by applying *mutation* and *crossover* to selected individuals. The following sections describe these mechanisms in more detail.

Selection is the process of choosing a subset of individuals from a population. This is done for two main purposes:

1. Selecting individuals for genetic operations such as mutation or crossover.
2. Selecting individuals to persist unchanged into the next generation.

Only keeping the best individuals (also referred to as *truncation selection* is a very simple method. It quickly removes weak individuals, but also results in low genetic diversity and hence can easily get stuck in local optima. Another simple selection algorithm is random selection, with the probability that an individual is selected being based on its comparative fitness. A particular implementation is *roulette wheel* or *fitness proportionate selection*. A more advanced selection method that was used in our algorithm is *tournament selection* [21]. In each "tournament" round, n individuals are selected randomly from the whole population. Among these n individuals, the one with the highest fitness wins and becomes part of the offspring. This process is repeated until the desired number of offspring individuals is reached.

The number of individuals for each round (n) has a direct influence on the probability that a weak individual (low fitness) makes it into the offspring. The smaller n is, the more likely it is that none of the top individuals are contained in this random selection. This can be described as *selection intensity*. A higher selection intensity will remove weaker individuals much faster [8].

Crossover is also called *sexual recombination* [6]. The genetic information of two parent individuals is merged to create one or two offspring individuals. For an N-Point crossover, the parents' genetic strings are cut at N points, creating N+1 segments of genetic information. For the first offspring individual, each segment is randomly taken from either one or the other parent. The remaining segments can create a second, "inverted" offspring or be discarded.

Mutation introduces random changes into an individual. It is a mechanism that helps to prevent premature convergence[1]. Mutation helps explore regions in the search space that could not be reached before. The higher the probability of a mutation to occur, the more the algorithm will start to resemble a random search. Goldberg [3] suggests that this starts getting visible at a probability of 0.1 and that 0.5 means a de-facto random search. He finds that one mutation per 1000 bits obtains good results in empirical studies and that mutation rates are equal or smaller in natural populations. Compared to crossover, mutation can therefore be regarded a secondary mechanism.

3 Design and Implementation

In this chapter we describe the design and implementation of our solution to evolve good search control heuristics using genetic algorithms. Our system relies on DEAP [2] as the overall framework, and instantiates this framework with our representation of individuals and appropriate realizations of the genetic operators.

3.1 Algorithm overview

DEAP (Distributed Evolutionary Algorithms in Python) is a Python-based evolutionary computation framework. We chose it because it is current, well-supported, open source, but also because it is designed to be transparent, as opposed to a black-box approach hiding the actual functionality.

We do not have *the* genetic algorithm that generates new search heuristics, but instead different sub-algorithms or *genetic modules* that can be combined to create a particular algorithm. The DEAP framework proved useful for maintaining a logical code structure and for providing sub-algorithms such as for selection, thus speeding up prototyping.

Each particular algorithm instance is specified in a JSON-file. The file contains general parameters such as the initial population size or the problem set to test on, and a list of genetic modules to be employed. A genetic module consists of a selection algorithm and a genetic operator to be used. For each of them, there are several parameters that can be customized. Thus a vast range of possible combinations exists.

The diagram 2 shows the basic procedure of our genetic algorithm. At first, we can either create a new population or load it from file. The latter is useful if we want to start with successful individuals or compare the effects certain parameters have. For each individual in the population, the fitness needs to be determined.

There are two loops, an outer and an inner one.

The outer loop is repeated for every generation until the pre-defined maximum number of generations (which is usually between 100 and 1000) is reached. At termination, several statistics and the final population is saved to the log file.

The inner loop iterates over all genetic modules. Each genetic module consists of selection and either mutation or crossover. The DEAP framework already provides a rich set of selection algorithms, which so far have been sufficient for our experiments. These include tournament and random selection, as well as simply choosing the N best individuals. Selection can be with or without replacement and based on percentage or absolute numbers. Mutation or crossover, as described in the next section, is performed on the individuals.

[1] In this context, convergence means reaching a maximal fitness value, with the whole population concentrated near this solution

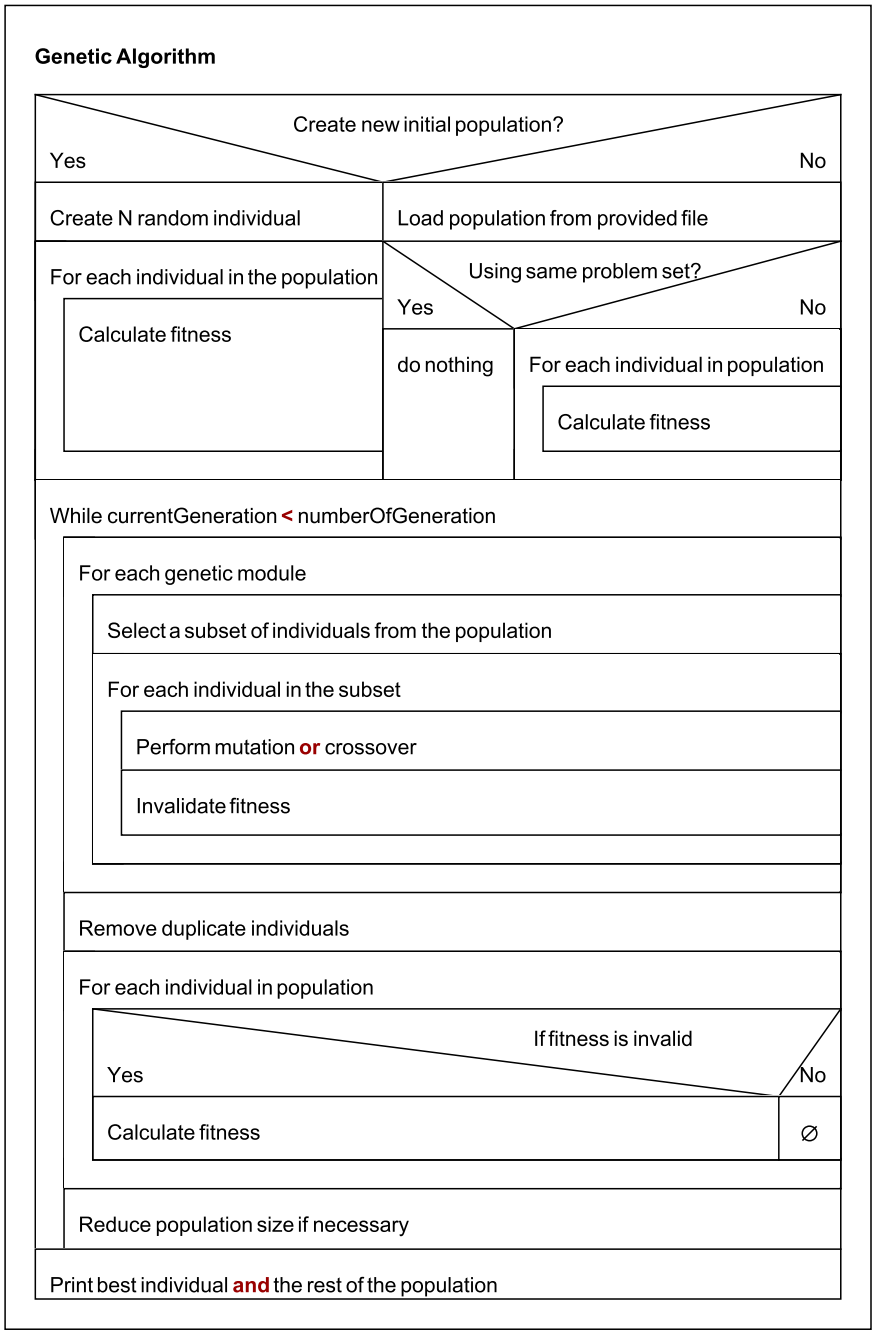

Figure 2: Basic algorithm flow

Back in the outer loop, duplicate individuals are removed. The frequency of these occurring depends very much on the parameters set for mutation and crossover. Also, at the end of each pass of the outer loop, the population size is reduced back to the desired number, since the individual genetic modules can, in sum, generate more individuals than desired. We can, e.g. implement three genetic modules which each select 40% of the population with replacement. Thus, without this reduction step, the population would grow exponentially, adding 20% with each generation. To prevent this, the population can be reduced to its original size by one or multiple addition selection algorithms. If selectBest() is used we get a very strong form of *elitism*.

3.2 Encoding of individuals

Individuals in the population correspond to search heuristics for the theorem prover. These search heuristics need to be represented by a Python data structure. The simplest way for an easy integration with DEAP is to represent each individual (or heuristic) as a list of weight functions. Each element of that list is a itself a list and represents a single weight function. It contains the function name (as used by E), the round-robin weight (used to determine how many clauses are selected according to this weight function), and all parameters required for the particular weight function in their correct order. A Python encoding of the search heurisitic from figure 1 is given below.

```
individual = [
   ['ConjectureRelativeSymbolWeight', 1, 'SimulateSOS',
       0.5, 100, 100, 100, 100, 1.5, 1.5, 1],
   ['ConjectureRelativeSymbolWeight', 4, 'ConstPrio',
      0.1, 100, 100, 100, 100, 1.5, 1.5, 1.5],
   ['FIFOWeight', 1, 'PreferProcessed'],
   ['ConjectureRelativeSymbolWeight', 1, 'PreferNonGoals',
      0.5, 100, 100, 100, 100, 1.5, 1.5, 1],
   ['Refinedweight', 4, 'SimulateSOS', 3, 2, 2, 1.5, 2]
]
```

When a new random individual for the initial population is created, the first random selection is which weight function from a dictionary of possible ones is used. Once the weight function is selected, a random value is chosen for each parameter the function requires. In the case of numeric values, the random value will be within a specified range. The limits of this range are specific for each parameter and set such that they are within an order of magnitude of successful human-designed individuals. The whole process is repeated until the individual consists of n different weight functions.[2]

3.3 Fitness

The quality of an individual strategy for E depends on two main properties: the number of problems that can be solved using this heuristic, and the amount of time needed for this. For simplicity, we use a fixed set of problems with a comparatively short time limit, and optimize only for the number of problems solved. Therefore the fitness of an individual is represented

[2] In our experiments, we usually chose n to be between 3 and 12. While a weight function may occur more than once within an individual, the probability that all parameters are identical is practically zero.

by an integer specifying the number of problems it solves. Obviously we want this value to be maximized.

In the concrete case, two parameters control test problem selection: the size of the problem set and the difficulty of the problems selected. We picked problems of medium difficulty, so that between 30 and 70 percent of them can be solved. A higher number of problems ensures that changes in the heuristics' qualities trigger changes in the (integer) number of solved problems. However, algorithm run-time also increases linearly with the number of problems (given that additional ones are of the same average difficulty as the previous average). The task is therefore to find a good balance between total run-time and a sufficiently granular step size.

We use problems from the TPTP library [16, 17], a large collection of standard benchmark problems for automated reasoning systems. TPTP problems have a difficulty rating, based on the performance of state-of-the-art provers on this problem. A rating of 1.0 means that currently no theorem prover taking part in the CASC competition can solve the problem, whereas a rating of 0 means that all of them can [18]. For our algorithm we chose random problems with a difficulty in the range of 0.25 and 0.4, thus ensuring that the aforementioned 30 to 70 percent are successfully solved. The randomly chosen set is saved to disk so that the same one will always be used after the initial random selection. This ensures that ratings of different runs are consistent and comparable.

3.4 Mutation

The mutation operator is initially quite simple. Its functionality is represented by the following pseudo code:

```
Input: individual, probWeightMutates, probParamMutates
Output: individual

for weight in individual:
    if random(0,1) < probWeightMutates:
        for p in weight: #p = parameter
            if random(0,1) < probParamMutates:
                if parameter.type == string:
                    paramNew = weightsDictionary.getRandomString()
                elsif parametert.type == int or float:
                    paramNew = random.gauss(mu=p, sigma=(max-min)/4)
                    if paramNew < 0 : paramNew = 0
                p = paramNew
```

Apart from the individual to be mutated, there are two additional input parameters: the probability that a single weight function mutates and the probability that a parameter within this weight function mutates. Most parameters are numerical (and real-valued), but the priority function parameter can be selected from a finite set.

One configuration we used for our initial evaluation is to set $probWeightMutates = 0.5$ and $probParamMutates = 0.3$. This means that if a heuristic is selected for mutation (which in itself is not very likely), the probability that a single parameter will mutate is $0.5 * 0.3 = 0.15 = 15\%$. The low probability of a parameter to mutate is one factor to ensure that new mutated offspring does not become too dissimilar from its parent. Another means for ensuring this similarity is the function for determining the new value. For numeric values, we use a normal distribution. The expected value μ is set to the old value of the parameter and the standard deviation σ is set to $1/4 * (max - min)$, where min and max are the reasonable limits we assumed in the

weight function dictionary. So with a probability of approximately 70%, the new value will not deviate more than σ from the old. However, more extreme mutations are not impossible, they are just less likely.

In the case of string parameters, we perform an equal-weight random selection from the set of possible values.

3.5 Crossover

We currently implement crossover both within individual weight functions and exchanging whole weight functions. The implementation of the crossover algorithm is visualized by this code fragment:

```
Input: individual1, individual2, favor1
Output: newIndividual

newInd = [] #this is what we return

#find matches among weight-functions
matches = []
for i1, e1 in enumerate(ind1):
    for i2, e2 in enumerate(ind2):
        if e1[0] == e2[0]:
            matches.append([e1,e2])
            individual1.remove(e1)
            individual2.remove(e2)
            break

nonmatches = individual1
nonmatches.extend(individual2) #get the remaining ones
```

The input for the crossover function are the two individuals on which this is to be performed and the bias (or favor) for the first individual. This parameter should be significantly larger than 0 and smaller than 1; otherwise we do not get any mixing of the individuals.

In the majority of experiments we used a Gaussian distribution centered around 0.5 with a standard deviation of 0.25. In this crossover algorithm we do not know which of the two has a higher fitness. Therefore it makes no sense to specifically favor either of them. Gaussian distribution ensures a greater variety of combination, as we do not just get a 50:50 selection of parameters. Whether there is a true advantage to this is to be determined by further experiments.

We can only perform in-function crossover of two individuals on weight functions contained in both of them. When there are matching ones found (simply by comparing the name), these are put into the matching list. The rest of the procedure ensures that we have all non-matching ones in another list.

```
def cxParams(w1, w2, favor1):
    newInd = []
    for w1, w2 in zip(w1, w2):
        if random.uniform(0,1) < favor1:
               newInd.append(w1)
        else:
               newInd.append(w2)
```

```
    return newInd

for m in matches:
    newInd.append(cxParams(m[0], m[1], favor1))
```

In the next step, we define a function that joins two weight function (i.e. their parameters) into a new one. It takes the two weights and the bias for the first weight as an input. Walking through all the parameters, we use either the one from the first or from the second weight function. This is done for elements in our matches list. Now the number of matches may be small or even zero. For this reason we may have to "fill" the heuristic with additional weight functions until a sufficient number is reached:

```
minNum = random.randint(MINW,MAXW)
while len(newInd) < minNum and len(nonmatches) > 0:
    new = random.choice(nonmatches)
    newInd.append(new)
    nonmatches.remove(new)

return newInd
```

To determine the number of weight functions the new heuristic should contain, we use the same values that are used for the initial creation of the population. In this example, the selection of additional weight functions is purely random. It would be possible to use the favor again here. Whether this would have a positive effect is again to be tested.

4 Experimental Evaluation

We performed experiments for two purposes: the main goal was to find better heuristics for E. To facilitate this, however, the parameters of the algorithm needed to be tuned. Large-scale runs with big populations and a high number of problems are very time intensive. To use the available resources more efficiently, we aimed to improve the algorithm itself and the parameters provided to it with smaller runs.

In the first stages, experiments were run such that crossover was performed among the best individuals and mutation for randomly selected individuals and a few of the best. Those individuals were then added to the original population. In the last step of each generation, the population was truncated to the original size by discarding the weakest individuals. This means a high selection intensity, which, on the one hand, has the disadvantage of reducing the diversity of the population, but on the other hand yields quite strong individuals in a short time.

For the selected set of proof problems to be sufficiently representative, we set the minimum amount to be used in actual optimization to 50. Sometimes up to 200 were used as well, but beyond that the total run-time of the algorithm time just gets too high for practical purposes. The duration of a single solving attempt is limited by CPU time. If no solution is found after 3 seconds, the problem is regarded as unsolved. It follows that the maximum execution time is n * t, with n being the number of problems and $t = 3sec$. However, many problems are solved within a fraction of a second, so the actual time used is less.

Figure 3 shows the result of one evolution experiment, which used a population size of 750 and 150 problems for fitness estimation, with both mutation and crossover enabled.

We have also compared one of the best "bred" clause selection heuristic (with identical

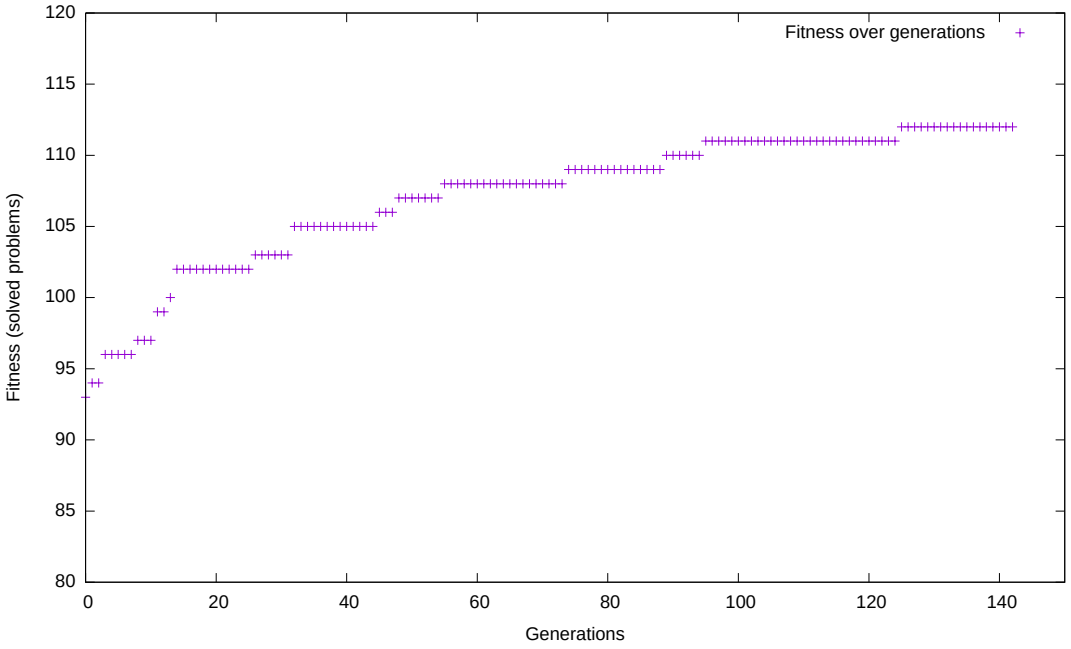

Figure 3: Fitness of the best member of the population over time

settings for the other parameters) to the best single strategy so far created manually. We ran the prover over all 15758 first-order problems from the TPTP library [16], version 6.0.0, with a time limit of 30 seconds on 2.6Ghz Intel machines. The best manual strategy found 8750 solutions with this time limit, the best bred strategy solved 8814, a modest but welcome increase. In particular, the evolved strategy solved 466 problems not solved by the designed one, and the designed one solved 386 not solved by the evolved one. Looking at the difference by TPTP domain, there are few obvious trends. The evolved strategy performs relativelty well in CSR (*Common Sense Reasoning*) and GRP (*Group Theory*).

Figure 4 shows the evolved weight function. Without going into too much detail, we can point out a few interesting features. First, the evolved function also interleaves size- and age based selection, with a ration of 14-1 as compared with 10-1 for the designed function. Secondly, the preference for goal symbols (the first numerical parameter in `ConjectureRelativeSymbolWeight`) is much more aggressive in the evolved example. And third, many of the other parameters are in a similar range in both cases.

5 Conclusion

We have presented a framework and a first implementation for applying genetic algorithms to the hard problem of finding good search heuristics for automated theorem provers. Our first results show that even a simple approach can automatically find heuristics that match, and in some cases exceed, the performance of heuristics that have been semi-manually tuned by developers, sometimes over the span of many years.

There remain a number of open points. First, our evaluation is quite preliminary. We

```
(1*ConjectureGeneralSymbolWeight(SimulateSOS,
      488,104,105,32,173,0,327,3.63456933832,1.43436931217,1),
 1*FIFOWeight(PreferProcessed),
 8*Clauseweight(PreferUnitGroundGoals,
      1,1,0.492432663985),
 3*Refinedweight(PreferGoals,
      2,4,7,4.94895515161,6.55154057122),
 2*ConjectureRelativeSymbolWeight(Constprio,
      0.0640406356983,67,160,111,25,3.13817216288,2.75641029012,1))'
```

Figure 4: Example of a good evolved evaluation heuristic

will conduct additional experiments in the genetic phase (growing new strategies), and also perform a more thorough comparison of bred and designed strategies. To facilitate large-scale experimenting, we are looking at the option to move evaluation to low-cost cloud providers, e.g. Google Preemptive Instances, which offer computing power for low-critically jobs at very competitive rates even compared to fully utilized in-house servers.

In addition to improved evaluation, we will open the ecosystem in two ways. First, we will broaden the number of options accessible to the prover, and in particular also include term ordering and literal selection strategies in the genome. Secondly, we will experiment with different fitness functions, in particular by favouring strategies that solve instances that are solved by few other strategies. Thus, we hope to breed ensembles of strategies that would be useful in strategy-scheduled or strategy-parallel systems.

References

[1] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[2] Fèlix-Antoine Fortin, François-Michel De Rainville, Marc-Andrè Gardner, Marc Parizeau, and Christian Gagnè. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[3] D.E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, 1989.

[4] Kryštof Hoder and Andrei Voronkov. The 481 ways to split a clause and deal with propositional variables. In Maria Paola Bonacina, editor, *Proc. of the 24th CADE, Lake Placid*, volume 7898 of *LNAI*, pages 450–464. Springer, 2013.

[5] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Proc. of the 25th CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[6] Jorge Magalhães-Mendes. A comparative study of crossover operators for genetic algorithms to solve the job shop scheduling problem. *WSEAS transactions on computers*, 12(4):164–173, 2013.

[7] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science and MIT Press, 2001.

[8] Hartmut Pohlheim. GEATbx: Genetic and evolutionary algorithm toolbox for use with MATLAB documentation - section 3 (selection). `http://www.geatbx.com/docu/algindex-02.html#P533_26754`, 2006.

[9] A. Riazanov and A. Voronkov. The Design and Implementation of VAMPIRE. *Journal of AI Communications*, 15(2/3):91–110, 2002.

[10] S. Schulz. Learning Search Control Knowledge for Equational Theorem Proving. In F. Baader, G. Brewka, and T. Eiter, editors, *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 320–334. Springer, 2001.

[11] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[12] Stephan Schulz. Fingerprint Indexing for Paramodulation and Rewriting. In Bernhard Gramlich, Ulrike Sattler, and Dale Miller, editors, *Proc. of the 6th IJCAR, Manchester*, volume 7364 of *LNAI*, pages 477–483. Springer, 2012.

[13] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *LNAI*, pages 45–67. Springer, 2013.

[14] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[15] R. Sekar, I.V. Ramakrishnan, and A. Voronkov. Term Indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1961. Elsevier Science and MIT Press, 2001.

[16] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[17] Geoff Sutcliffe. The TPTP World - infrastructure for automated reasoning. In E. Clarke and A. Voronkov, editors, *Proc. of the 16th LPAR, Dakar*, number 6355 in LNAI, pages 1–12. Springer, 2010.

[18] Geoff Sutcliffe. The TPTP problem library - TPTP v6.0.0. Technical report, Department of Computer Science, University of Miami, Miami, FL, 2014. `http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml`.

[19] Josef Urban. Preprint arXiv:1301.2683 [cs.AI], 2014.

[20] Christoph Weidenbach, Renate Schmidt, Thomas Hillenbrand, Dalibor Topić, and Rostislav Rusev. SPASS Version 3.0. In Frank Pfenning, editor, *Proc. of the 21st CADE, Bremen*, volume 4603 of *LNAI*, pages 514–520. Springer, 2007.

[21] Jinghui Zhong, Xiaomin Hu, Min Gu, and Jun Zhang. Comparison of performance between different selection strategies on simple genetic algorithms. In *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, volume 2, pages 1115–1121, Nov 2005.