This is a preprint copy of:

Klein, J., and L. Spector. 2008. Genetic Programming with Historically Assessed Hardness. In *Genetic Programming Theory and Practice VI*, edited by R. L. Riolo, T. Soule, and B. Worzel. New York: Springer-Verlag. In press.

NOTE: The chapter and page numbers on this copy are not final.

Chapter 1

GENETIC PROGRAMMING WITH HISTORICALLY ASSESSED HARDNESS

Jon Klein[1] and Lee Spector[1]

[1] *Cognitive Science, Hampshire College, Amherst, MA USA 01002-3359*

Abstract

We present a variation of the genetic programming algorithm, called Historically Assessed Hardness (HAH), in which the fitness rewards for particular test cases are scaled in proportion to their relative difficulty as gauged by historical solution rates. The method is similar in many respects to some realizations of techniques such as implicit fitness sharing, stepwise adaptation of weights and fitness case selection, but the details differ and HAH is generally simpler and more efficient. It also leads to different generalizations. We present results from large-scale, systematic tests of HAH and we also discuss the technique in terms of the alternative explanations that it supports for the efficacy of implicit fitness sharing and related methods.

Keywords: historically assessed hardness, implicit fitness sharing, Push, PushGP

1. Introduction

Some problems are harder than others. Within the context of a single problem, some test cases are generally harder than others. Good performance on the hard cases is probably more valuable, all other things being equal, than good performance on the easy cases. How can this differential value be incorporated into the reward function of a machine learning system? Will the incorporation of differential rewards based on problem/case difficulty improve the system's learning performance?

In this chapter we address these questions in the context of genetic programming (Koza, 1992), in which the reward function is called a "fitness function" and test cases are called "fitness cases." We provide a simple mechanism for giving greater fitness rewards for good performance on harder fitness cases, and we describe the mechanism's impact on performance and relation to other techniques.

"Hardness," however, is not itself a simple concept. Some of the hardness of a fitness case may be intrinsic to the problem that we are trying to solve, or at least to the relation between the problem and the problem-solving tools provided by the genetic programming system. For example, suppose that we are working on a symbolic regression problem, trying to discover a formula that explains a data set, and that the correct answer has a form like:

$$y = \begin{cases} x + 0.4 \text{ if } x > \pi \\ x - 0.4 \text{ if } x < -\pi \\ 1.07x^{11} - 11x^{\pi} + \pi x \text{ if } -\pi \leq x \leq \pi \end{cases}$$

Clearly there is a sense in which the cases with x values between $-\pi$ and π are harder, but is this really an intrinsic property of the problem? Suppose that the genetic programming function set includes standard arithmetic operators, a conditional control structure, and a general exponentiation function, but that the available constants are just those in the set:

$$\{\pi, -\pi, 1.07, 11\}$$

Note that this set does not include the constant 0.4 or "ephemeral random constants" that could produce 0.4. Furthermore, while it is possible to construct 0.4 using the provided constants and standard arithmetic operators it is not trivial to do so. It therefore seems likely that it will actually be *easier* to produce formulae for cases with x values between $-\pi$ and π. So at least some of the hardness of a fitness case may be due to the match between the problem and the functions and constants that are available to the genetic programming system. Whether or not there is a meaningful sense in which fitness cases have "intrinsic" hardness independent of these factors is a complex question that we

will not address here, but clearly there are components of hardness that derive from the conjunction of particular problems with particular constraints on the elements out of which solutions can be constructed.

Other aspects of fitness case hardness will derive from the implementation choices made by the user, including the choice of program representation (for example, whether programs are represented as trees, linear strings, grammatical derivations, etc.), the choice of genetic operators (including various types of mutation and crossover), and choices of parameters (including the proportion of each generation produced by each of the chosen genetic operators). These choices collectively define the ways in which a population of programs can be transformed from generation to generation. In combination with the fitness function they define the fitness landscape that genetic programming can search. On some such landscapes programs that perform correctly on particular fitness cases will be more or less accessible than they are on other such landscapes; therefore, within the context of particular implementation choices some fitness cases will be harder or less hard than they are in the context of other implementation choices.

A final meaning of hardness grows out of a population's current distribution on a fitness landscape, which in turn determines how easily the population can move towards a solution. For example, it is widely accepted that genetic diverisity is a beneficial trait of a genetic programming population because a diverse population is likely to achieve a more thorough exploration of the fitness landscape. Likewise, a population that is mostly centered around steep local maxima is likely to perform worse than a population of equal (or even worse) average fitness that is positioned better relative to the desired solution. This form of hardness is not intrinsic to a problem and changes dramatically across and even during genetic programming runs. Because of its unpredictability and dynamic nature, this form of hardness may be difficult to counteract using non-dynamic strategies such as changing a problem's representation or modifying a run's parameters.

Hardness based on the population distribution derives in large part from "choices" made by the random code generator at the start of a genetic programming run and from other partially-random events such as choices of crossover points and new code produced by mutation. Such events can affect the hardness of particular fitness cases in several ways. For example, if particular functions or constants that are needed for some cases are rare in the initial population of a run then those cases may appear to be harder in the context of that run. Similarly, an initial population that contains a large number of individuals that do well on a particular subset of the fitness cases may lead to subsequent generations dominated by individuals that "specialize" in these cases, making progress on the other cases more difficult.

Our goal in the present project is to develop a method that focuses a genetic programming system's effort on the hard cases, regardless of the nature or source of the underlying hardness. We want to give more rewards for doing well on the hard stuff—for performing better on the more challenging fitness cases—regardless of whether the hardness is due to intrinsic difficulty, representation and implementation decisions, historical "accidents" in a particular run, or combinations of these and other factors. We have accomplished this by developing a technique in which hardness is gauged historically, on the basis of solution rates in the current run. This is easy to do and it captures, in some sense, all of the sources of hardness described above. As we show below, it also seems to produce significant improvements in genetic programming performance, at least in the contexts in which we have tested it.

Some aspects of our new technique, when viewed at the implementation level, are similar to aspects of other techniques such as "implicit fitness sharing" (sometimes also called "phenotypic fitness sharing"), "stepwise adaptation of weights," and various methods for selecting fitness cases. But these techniques all differ from our new technique both conceptually and with respect to their potential extensions. We discuss the relations between these techniques below, after the presentation of our new method.

2. Historically Assessed Hardness

Our new method, genetic programming with Historically Assessed Hardness (HAH), involves the scaling of values for individual fitness cases based on their empirically determined difficulty. As in much of the literature, we phrase our discussion here in terms of fitness values that are based on errors, and lower values (derived from smaller errors) are better than larger values (derived from higher errors). In this context, most traditional genetic programming systems compute the overall fitness F_i of individual i for n fitness cases as the sum of the absolute value of individual error $|e|$ for each case c; we call this the "unscaled summation" method:

$$F_i = \sum_{c=0}^{n} |e_c|$$

In HAH we differentially scale the individual terms in this summation, and there several ways in which this can be done. In all of the methods that we have studied the scaling function involves a historically assessed *solution rate* for the individual case, designated here as s_c. This solution rate can itself be calculated in several ways; those that we have tested are:

Previous Generation s_c is the fraction of the individuals in the previous generation that produced the correct answer for fitness case c.

Current Generation s_c is the fraction of the individuals in the current generation that produce the correct answer for fitness case c. Note that this requires that the scaling be performed in a second pass through the current generation, after errors have been calculated for all individuals. Because of the extra pass required, we do not advocate the use of this method, but we include it in our experiments for the sake of comparison against implicit fitness sharing.

Cumulative s_c is the fraction of all previously-evaluated individuals in the current run that produced the correct answer for fitness case c. We base this on the errors of all individuals in all previous generations of the current run.

Cross-Run s_c is the fraction of all individuals in a previous, independent run of genetic programming that produced the correct answer for fitness case c. The independent run (from which the rates are derived) is conducted using the unscaled summation (standard) method for fitness case combination.

However the solution rates (s_c values) are calculated they can be used in a variety of ways to scale the individual errors for each case. We have explored two such methods:

Difference In this method the solution rates, which run from 0 to 1, are subtracted from 1.01 and the product of this difference and the raw error value is used as the scaled value:

$$F_i = \sum_{c=0}^{n} (1.01 - s_c)|e_c|$$

The difference is calculated from 1.01 rather than 1.0 to avoid producing a scaled error of 0, which would be interpreted as success in the rare (but possible) circumstance that e_c is non-zero and s_c is 1.

Quotient In this method the solution rate is multiplied by the population size and divided into the the raw error:

$$F_i = \sum_{c=0}^{n} \frac{|e_c|}{1 + (s_c * P)}$$

The "$+1$" in the denominator prevents division by zero.

In both of these methods the error for each fitness case is scaled in a way that makes it larger when the solution rate for the case is smaller and smaller when the solution rate for the case is larger. In other words, errors count more (and therefore good performance is rewarded more) for cases that are harder.

3. Related Techniques

Historicallly Assessed Hardness is similar in many respects to a number of existing and well-studied techniques in evolutionary computation, including implicit fitness sharing, stepwise adaptation of weights, and other methods for the selection or weighting of fitness cases such as boosting and bagging (Iba, 1999). Given certain parameters, constraints and problem types, HAH may even be identical to some realizations of these methods.

Broadly stated, what these methods have in common is the scaling of fitness or error values for individual fitness cases based on some heuristic. The details of the methods differ: the scaling values may be computed dynamically at each generation, or at discrete points within a run; the scaling values may be continuous, or they may be binary (e.g. some techniques for fitness case selection can be described as using a scaling of 0.0 for non-active fitness cases); the scaling values may be computed based on a population's current performance, or on other heuristics.

A full comparison of HAH to all related methods is beyond the scope of this chapter, but we do compare it to a few of its closest relatives in the following sub-sections. For these, and for the others of which we are aware, the bottom line appears to be that while there are overlaps among the techniques each leads to different generalizations and HAH is the simplest method to implement.

Implicit Fitness Sharing

Fitness sharing is a technique for improving the performance of evolutionary computation systems, ostensibly by promoting diversity among the evolving population. In the original formulations of fitness sharing (Deb and Goldberg, 1989) fitness payoffs were reduced in densely populated regions of the search space by performing genotypic analysis and by sharing a local fitness budget among genotypically similar individuals (which collectively form a "niche"). This approach explicitly addresses population diversity at the expense of potentially costly genotype analysis to determine the niches (potentially $O(N^2)$ (Sareni and Krahenbuhl, 1998)). Further refinements of fitness sharing address concerns raised by the difficulty of genotypic analysis by assigning niches based on *phenotypic* analysis of the similarity of fitness profiles over all test cases. Phenotypic analysis may lead to simpler pairwise analysis of individuals, but it still leads to combinatorial problems in determining niches.

Implicit fitness sharing (Smith et al., 1993; McKay, 2001) foregoes the explicit computation of genotypic or phenotypic niches by divvying up fitness rewards for individual fitness cases based on the number of individuals that produce the same prediction. Although this avoids the need for the explicit computation of differences between individuals, it does require population-wide statistics to be calculated in a multi-pass fashion: the number of indi-

viduals sharing the fitness reward is not known until all individuals have been evaluated, so a second pass must be made through the population to compute fitness values. In the terminology of Section 2, implicit fitness sharing, at least when applied to Boolean problems, is nearly equivalent to HAH with error rates determined from the current generation and individual errors combined using the quotient method. Note, however, that this is a relatively complicated version of HAH to implement because it requires multiple passes through the population in order to make use of the statistics of the current generation. The other versions of HAH all differ from fitness sharing at the implementation level and are all simpler and more efficient to implement than implicit fitness sharing. In fact, when the "cross-run" version of the technique is used it is not necessary to compute any run-time population statistics at all, and although this version is far less effective than the others it nonetheless sometimes provides a benefit over traditional genetic programming.

The differences in the implementations of HAH and implicit fitness sharing stem from the fundamental differences in the theoretical underpinnings of the techniques. Fitness sharing was motivated by a desire to maintain population diversity, while HAH is intended to focus search on the difficult parts of a problem. Sometimes these goals overlap, and in some cases both goals may be served by the same modification to the standard genetic programming technique. But even in such cases the differences in theoretical underpinnings lead to different ideas for extension and generalization and to different explanations of the successes or failures of the techniques in particular circumstances.

A somewhat subtle difference between HAH and implicit fitness sharing, which nonetheless has practical implications, concerns the way in which fitness is shared between individuals. HAH scales fitness rewards/penalties according to the proportion of individuals that produce the correct answer, while fitness sharing divides fitness values among all individuals with the *same* answer, correct or incorrect. For problems with Boolean outputs, in which all incorrect answers for a particular fitness case are necessarily identical, this distinction has no practical impact. For problems in which different incorrect answers may receive different fitness penalties, however, fitness sharing and HAH differ significantly. HAH, as defined above, will still scale all fitness penalties (including penalties for suboptimal solutions) by a case's overall solution rate, while fitness sharing will divide fitness rewards based on the number of individuals producing the same result. For individuals that belong to groups that do not produce the optimal answer, their scaling factor with fitness sharing will not be equal to the fitness case solution rate.

Another way to think about the difference between fitness sharing and HAH is that fitness sharing schemes typically treat fitness rewards as zero-sum resources that are divided up equally among all individuals that produce the same response. HAH differs philosophically from fitness sharing on this point: in

HAH fitness rewards or penalties are scaled in order to promote the solution of difficult problems, not to treat fitness as a zero-sum resource. Treating fitness as a zero-sum resource leads to an extreme focus on promoting population diversity, even at the expense of high-quality individuals. This can lead to pathological cases in which an individual that finds only a few correct responses can be awarded a better fitness than individuals that perform far better but also produce correct outputs on fitness cases that are solved by many other individuals. Because implicit fitness sharing does not use any actual (i.e., explicit) measure of similarity between individuals, there is no assurance that the individuals sharing fitness are actually similar, either genotypically or phenotypically (because the fitness reward sharing is done on a per-fitness-case basis, not accounting for an individual's entire phenotypic profile).

Stepwise Adaptation of Weights

Stepwise Adaptation of Weights (SAW) is a related technique which modifies a fitness landscape by scaling an individual's error values based on a set of evolving weights. The technique was originally applied to constraint satisfaction problems in which the weights were applied to individual constraints (Eiben and van Hemert, 1999), but has also been applied to genetic programming (and symbolic regression in particular) by applying the weights to individual fitness cases (Eggermont and van Hemert, 2000).

When applied to symbolic regression problems, SAW resembles HAH in that weights are modified according to performance on a set of fitness cases, but the details of how the weights are computed differ greatly between the techniques. In a typical SAW implementation, weight adjustments are performed according to the performance of the best individual in the population, and only once every several generations (5-20 in the work on symbolic regression, but up to 250 in the original work with constraint satisfaction problems). Weights are initialized to $w_i = 1$ and are updated by adding a constant value if the error for fitness case i is non-zero. An alternative weight updating scheme, dubbed precision SAW, updates the weights in proportion to the fitness case error value itself (Eggermont and van Hemert, 2000) as is done with HAH, although the weight changes in SAW are cumulative while in HAH the weights are recomputed at each evaluation.

Because there is a sense in which, given certain parameters and assumptions, HAH is similar to both implicit fitness sharing and SAW, it is reasonable to suggest that SAW can be considered a form of fitness sharing and vice versa. It seems at first glance counterintuitive that SAW, which uses only the best individual in a population, could be considered a special case of fitness sharing, which depends on measures of phenotypic or genotypic similarity between individuals. However, if one considers the notion that the genes of the best in-

dividuals in a population are more likely to be over-represented relative to less successful individuals, then using only the performance of the best individual in a population may be a reasonable (though greatly simplified) proxy measure of the dominant phenotypes in the population.

Fitness Case Selection

Another broad area of research related to HAH concerns fitness case selection methods. These use various heuristics for choosing subsets of the fitness cases for fitness testing. Many of these are intended to reduce the number of fitness cases so that fitness evaluations can be run more quickly, and many use subsets based on random selection. Dynamic Subset Selection (DSS) expands on the notion of fitness case selection by incorporating fitness case difficulty as a criterion for fitness case selection (Gathercole and Ross, 1994). More recently, Topology Based Selection (TBS), a heuristic based on similarity between fitness cases (based on co-incidence of their solution) has proven to be even more effective than DSS at improving the performance of GP on regression and classification problems (Lasarczyk et al., 2004).

HAH does not deal explicitly with the selection of fitness case subsets for testing, but fitness case selection-like effects do sometimes emerge with HAH, particularly when the population evolves to the point at which certain fitness cases are fully solved. However, the most notable benefit of fitness case selection methods, namely a reduction in the number of fitness cases, is not present in HAH. So while HAH, like some variations of fitness case selection, may improve the performance of GP by changing the fitness landscape, HAH is not useful in reducing computation time required for individual fitness evaluations.

4. Experiments

We examined the performance of Historically Assessed Hardness in the evolution of a solution to the n-parity problem, in which a program must determine whether a variable-length list of inputs has an even or odd number of "true" values. We chose this problem, which we have studied previously, as an example of a moderately difficult classification problem.

We used the PushGP genetic programming environment (Spector, 2001; Spector and Robinson, 2002; Spector et al., 2004; Spector et al., 2005) integrated with the breve simulation environment (Klein, 2002). Push is a multi-type stack-based programming language developed for evolutionary computation and which allows the evolution of novel control structures through explicit code and control manipulation. These control structures are of particular interest for problems such as n-parity which require iterative or recursive solutions, especially in the absence of explicit iterator functions, as in the current work.

Table 1-1. Instruction set for tests of Historically Assessed Hardness on the n-parity problem. Full documentation for these instructions is available in the Push language specification (Spector et al., 2004).

Types	Instructions
Integer instructions	FROMBOOLEAN $<$ $>$ MAX MIN % / * - + STACKDEPTH SHOVE YANKDUP YANK = FLUSH ROT SWAP POP DUP RAND
Boolean instructions	FROMINTEGER NOT OR AND STACKDEPTH SHOVE YANKDUP YANK = FLUSH ROT SWAP POP DUP
Code instructions	FROMNAME FROMBOOLEAN FROMINTEGER DO SIZE LENGTH EXTRACT INSERT NTHCDR NTH APPEND LIST NOOP IF DO* CONS CDR CAR NULL ATOM QUOTE STACKDEPTH SHOVE YANKDUP YANK = FLUSH ROT SWAP POP DUP
Name instructions	QUOTE STACKDEPTH SHOVE YANKDUP YANK = FLUSH ROT SWAP POP DUP
Exec instructions	Y S K IF STACKDEPTH FLUSH POP DUP

Table 1-2. Parameters used with PushGP for the n-parity problem runs.

Population size	5000
Evaluation limit	2000
Mutation	40% fair mutation 5% deletion mutation
Crossover	40%
Copy operator	15%
Tournament selection size	7
Generation limit	300
Fitness cases	64 randomly generated lists of between 8 and 12 Boolean values (the same randomly generated fitness cases were used for all runs)

The instruction set used is found in Table 1-1. The instruction set includes standard stack manipulation instructions for all types, integer math instructions, Boolean logic instructions and instructions for explicit manipulation of code and program control (via the EXEC stack, which contains a list of all instructions to be executed).

Notably absent from the instruction set we used are explicit iterator instructions that Push supports. These iterator functions (such as CODE.DOTIMES and EXEC.DO*COUNT, among others) allow for the construction of explicit

Table 1-3. Results of various fitness scaling methods on the n-parity problem, using the parameters in Table 1-2 and the instructions in Table 1-1, sorted by success rate ("% Succ."). Abbreviations used in the method descriptions: Prev = Previous Generation, Cum = Cumulative, Cross = Cross-Run, Curr = Current Generation, Diff = Difference, Quot = Quotient. Note that "Curr/Quot" is equivalent to implicit fitness sharing for the n-parity problem (because it is a Boolean problem; see text). Computational effort, described by Koza, represents the cumulative effort required to solve a problem, taking into account both solution rate and generations required to find a solution (Koza, 1992).

Method	Succ. /Runs	**% Succ.**	Av. Succ. Gen.	Comp. Effort	Mean Best Fitness
Prev/Diff	180/233	**77.253**	125.622	1,800,000	0.9674
Curr/Quot	169/234	**72.222**	111.520	1,752,000	0.9957
Prev/Quot	162/234	**69.230**	126.456	2,124,000	1.0206
Cum/Diff	157/232	**67.672**	129.713	2,208,000	1.0734
Cum/Quot	140/234	**59.829**	137.671	2,880,000	1.0721
Cross/Quot	57/234	**24.358**	157.140	9,288,000	1.5335
Cross/Diff	49/234	**20.940**	126.448	4,304,000	1.5678
Unscaled	48/234	**20.512**	120.395	8,282,000	1.6515

loops, and the absence of these instructions makes finding solutions considerably more difficult. In place of iterator functions are special combinator functions (Y, K and S) which manipulate the EXEC stack and which can be helpful in the evolution of iterative or recursive behaviors that use novel, evolved control structures (Spector et al., 2005).

A complete list of the parameters used during the experiment is shown in Table 1-2. We conducted 234 runs of each technique, with 4 runs lost to miscellaneous software and system problems.

5. Results

The results of our runs are shown in Table 1-3. The HAH configuration in which solution rates were obtained from the previous generation and errors were scaled using the difference method produced the most solutions overall and the lowest (best) mean best fitness. Several other configurations also performed quite well; in fact all of the HAH configurations aside from the cross-run configurations significantly out-performed the standard, unscaled method of combining errors on individual fitness cases. The savings provided by each of these methods, in terms of computational effort, was about 75%.

The configuration that is essentially equivalent to implicit fitness sharing ("Curr/Quot") was among the best; indeed it had the lowest average success generation and the lowest computational effort. But the implementation of this method (using the statistics of the current generation) is somewhat more

complex than that of the others, so it is not necessarily the best choice. It is also important to bear in mind that none of these methods is strictly equivalent to implicit fitness sharing when applied to non-Boolean problems.

The performance of the cross-run HAH methods, while much worse than that of the other HAH methods, is interesting in several respects. In terms of computational effort one of these methods ("Cross/Diff") performed much better than the standard ("Unscaled") method, but the other of these methods ("Cross/Quot") did somewhat worse than standard. However, both of these methods out-performed the standard method both in terms of overall success rate and in terms of mean best fitness. The high computational effort of "Cross/Quot" is attributable to the relatively large number of generations it required to achieve success. Hence there are several senses in which even these methods, which used solution rates from independent runs as the basis for their scaling of errors, out-perform the standard method. To the extent that this is true one can infer that the improvements are due to aspects of case "hardness" that transcend the circumstances of a particular genetic programming run.

6. Conclusions and Future Work

We have presented a family of easily implemented methods that are designed to focus the effort of a genetic programming run on harder fitness cases, where hardness is determined empirically from solution rates during a genetic programming run. We have shown that these methods can produce significant improvements in problem-solving performance for a small investment of programmer effort. One of our methods turns out to be nearly identical, at the implementation level, with one type of fitness sharing that has been described in the literature (implicit fitness sharing), but the rationale for our methods is completely different from that offered in discussions of fitness sharing. Whereas fitness sharing is focused on the maintenance of diversity in a population, our Historically Assessed Hardness (HAH) methods are based on the idea of rewarding good performance on difficult problems. This difference in rationale leads to variations and extensions of our technique that are different than those that arise naturally from fitness sharing approaches.

We believe that HAH can have significant utility in a wide range of application areas. In fact, the exploration documented in this chapter was launched because our use of a form of HAH helped us to achieve new results in an application-oriented project (Spector et al., 2008).[1] In that project, however, we did not conduct sufficiently many or sufficiently systematic runs to say anything definitive about the efficacy of the method. Here we have conducted

[1]The formulation used in the prior work was essentially the method that we have described here as "previous generation, difference scaling," although the scaled values ran from 1.0 to 2.0 rather than 0.01 to 1.01.

$1,872$ runs on a well-studied problem and shown that the method does clearly have utility.

One direction for further research concerns the application of HAH to non-Boolean problems. As described above, in the context of such problems HAH is less similar to fitness sharing and there are reasons to believe that it will also be useful here. But we cannot say anything definitive about this without large-scale, systematic testing.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0308540. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. Special thanks to Wolfgang Banzhaf, Michael Korns, Loryfel Nunez, Katya Vladislavleva, Guido Smits, Malcolm Heywood and Ian Lindsay.

References

Deb, Kalyanmoy and Goldberg, David E. (1989). An investigation of niche and species formation in genetic function optimization. In *Proceedings of the third international conference on Genetic algorithms*, pages 42–50, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Eggermont, J. and van Hemert, J. I. (2000). Stepwise adaptation of weights for symbolic regression with genetic programming. In *Proceedings of the Twelveth Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'00)*.

Eiben, Gusz and van Hemert, Jano (1999). SAW-ing EAs: Adapting the fitness function for solving constrained problems. In Corne, David, Dorigo, Marco, and Glover, Fred, editors, *New Ideas in Optimization*, pages 389–402. McGraw-Hill, London.

Gathercole, Chris and Ross, Peter (1994). Dynamic training subset selection for supervised learning in genetic programming. In Davidor, Yuval, Schwefel, Hans-Paul, and Manner, Reinhard, editors, *Parallel Problem Solving from Nature III*, volume 866, pages 312–321, Jerusalem. Springer-Verlag.

Iba, Hitoshi (1999). Bagging, boosting, and bloating in genetic programming. In Banzhaf, Wolfgang, Daida, Jason, Eiben, Agoston E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1053–1060, Orlando, Florida, USA. Morgan Kaufmann.

Klein, J. (2002). breve: a 3d environment for the simulation of decentralized systems and artificial life. In Standish, R., Bedau, M. A., and Abbass, H. A., editors, *Proc. Eighth Intl. Conf. on Artificial Life*, pages 329–334. Cambridge, MA: MIT Press.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Lasarczyk, Christian W. G., Dittrich, Peter W. G., and Banzhaf, Wolfgang W. G. (2004). Dynamic subset selection based on a fitness case topology. *Evol. Comput.*, 12(2):223–242.

McKay, R. I. (Bob) (2001). An investigation of fitness sharing in genetic programming. *The Australian Journal of Intelligent Information Processing Systems*, 7(1/2):43–51.

Sareni, Bruno and Krahenbuhl, Laurent (1998). Fitness sharing and niching methods revisited. *IEEE Trans. Evolutionary Computation*, 2(3):97–106.

Smith, R., Forrest, S., and Perelson, A. (1993). Searching for diverse, cooperative populations with genetic algorithms.

Spector, Lee (2001). Autoconstructive evolution: Push, pushGP, and pushpop. In Spector, Lee, Goodman, Erik D., Wu, Annie, Langdon, W. B., Voigt, Hans-Michael, Gen, Mitsuo, Sen, Sandip, Dorigo, Marco, Pezeshk, Shahram, Garzon, Max H., and Burke, Edmund, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA. Morgan Kaufmann.

Spector, Lee, Clark, David M., Lindsay, Ian, Barr, Bradford, and Klein, Jon (2008). Genetic programming for finite algebras. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, London. ACM Press. To appear.

Spector, Lee, Klein, Jon, and Keijzer, Maarten (2005). The push3 execution stack and the evolution of control. In Beyer, Hans-Georg, O'Reilly, Una-May, Arnold, Dirk V., Banzhaf, Wolfgang, Blum, Christian, Bonabeau, Eric W., Cantu-Paz, Erick, Dasgupta, Dipankar, Deb, Kalyanmoy, Foster, James A., de Jong, Edwin D., Lipson, Hod, Llora, Xavier, Mancoridis, Spiros, Pelikan, Martin, Raidl, Guenther R., Soule, Terence, Tyrrell, Andy M., Watson, Jean-Paul, and Zitzler, Eckart, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA. ACM Press.

Spector, Lee, Perry, Chris, Klein, Jon, and Keijzer, Maarten (2004). Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, School of Cognitive Science, Hampshire College, USA.

Spector, Lee and Robinson, Alan (2002). Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40.