Multi-type, Self-adaptive Genetic Programming as an Agent Creation Tool

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

Alan Robinson
Cognitive Science
Hampshire College
Amherst, MA 01002
Alan@HciDesign.com

To appear in *Proceedings of the Workshop on Evolutionary Computation for Multi-Agent Systems, ECOMAS-2002*, International Society for Genetic and Evolutionary Computation. 2002.

Multi-type, Self-adaptive Genetic Programming as an Agent Creation Tool

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

Alan Robinson
Cognitive Science
Hampshire College
Amherst, MA 01002
Alan@HciDesign.com

Abstract

This paper describes the application of multi-type, self-adaptive genetic programming techniques, implemented in the *PushGP* and *Pushpop* systems, to the automatic programming of multi-agent systems. It includes a brief case study of the application of PushGP to a transport network control problem and a demonstration of self-adaptive modularization in a dynamic environment that was developed by Van Belle and Ackley.

1 INTRODUCTION

Automatic programming technologies hold great promise for the production of software agents for complex, dynamic environments. Genetic programming techniques, in particular, can harness the power of natural selection to produce autonomous agents and communities of agents well suited to their niches, even in environments that are too complex or dynamic for detailed human analysis. In a genetic programming system solutions emerge from a continuous process of adaptive engagement with the environment; in some cases this can produce solutions where other methods fail. Furthermore, automated methods can be repeatedly run and varied, and the results of these runs can be analyzed to provide detailed information about the difficulty of the agents' tasks and the suitability of the methods for the task environments. The genetic programming literature already describes many experiments in which systems with agent-like properties have been automatically generated. However, a gap remains between the capabilities of current genetic programming technology and those that would be required to automatically generate robust software agents for real-world environments. The project described here seeks to narrow this gap and to produce broadly-applicable agent creation tools based on evolutionary computing techniques.

One way to make agent development more replicable and amenable to formal analysis is to automate the process so that controlled, repeatable experiments can be performed. Unfortunately, however, the successful application of genetic programming technology is a "black art," often requiring repeated experiments and sophisticated understanding of interactions among a genetic programming system's many parameters. One goal of the described project is therefore to reduce the configuration work that a practitioner must do to apply the technique to a new problem area. The primary approach is to fold parameters into the representation over which the evolutionary search is conducted—that is, to allow the same adaptive process of natural selection to control both the search for agent programs and the search through the space of system parameters. This approach, of using self-adaptation to control system parameters, has been studied previously in evolutionary computing, but it has generally been applied fairly narrowly, for example in the self-adaptation of mutation rates or other numerical parameters. In the present project self-adaptive mechanisms are also employed to automatically develop data representations, program representations, population structure, and reproduction strategies. When it is adaptive to do so the resulting system may evolve toward any of a broad range of genetic algorithm or genetic programming approaches, but the user need not specify the details of the strategy or combination of strategies in advance.

Agents must generally interact with a heterogeneous set of entities, each of which may use a specialized interface or language. Early genetic programming systems forced users to restrict all operations to a single data type to ensure the semantic validity of programs undergoing recombination and mutation. More recently, "strongly typed" genetic programming systems have been developed that relax this restriction,

allowing the generation of programs that manipulate a diverse set of data types (Montana, 1995). This capability is critical for the evolution of agents that must interact with diverse entities and refer to diverse data. The described project extends the idea of strongly typed genetic programming to a more general notion of "multi-type" genetic programming that has several advantages. First, it allows the arbitrary intermixing, without syntactic restrictions, of code that works with any data type or set of data types. Second, it incorporates code types that simplify the dynamic evolution of subroutines, control structures, and other program representations. Third, it leverages the extended type system to allow agent programs to contain their own reproduction procedures, thereby transferring the responsibility for reproductive strategies from the user (via global, human-set parameters) to the system itself (via natural selection operating on the agents themselves). The cumulative effect of these developments is that the user can specify a diverse set of primitives and related data types while simultaneously specifying relatively little in the way of system parameters or about the potential architecture of solution agents.

The core technologies developed in this project are the Push programming language for evolved programs, the PushGP genetic programming system, and the Pushpop autoconstructive evolution system. Descriptions of these technologies have recently been published elsewhere (Spector, 2001; Spector and Robinson, 2002; Crawford-Marks and Spector, 2002). In this paper we provide only a brief synopsis of the technologies, followed by descriptions of two recent experiments with PushGP. While these two experiments both involved relatively simple problem environments, we believe that they nonetheless provide more general lessons about the use of evolutionary computing for the production of agents.

2 TECHNOLOGIES

2.1 THE PUSH PROGRAMMING LANGUAGE

Push is a programming language designed to facilitate the evolution of complex programs. Push is intended to be the language within which evolved programs are expressed—a Push-based evolutionary computation system might itself be written in any language (e.g. Lisp, C, or Java), but it would search the space of Push programs and produce a Push program as its output. A detailed description of the Push language, including a language reference, can be found in (Spector and Robinson, 2002).

Push is a stack-based language, similar in some ways to FORTH (Salman, 1984). The use of stack-based languages for genetic programming has several historical precedents upon which Push builds; see (Bruce, 1997) for a survey. The defining feature of a stack-based language is that arguments are passed to instructions and results are returned from instructions via global data stacks. Push provides an independent stack for each data type and Push instructions pop arguments from and push results onto whichever stacks are appropriate. This allows for the use of multiple data types in conjunction with complete syntactic uniformity—every possible sequence of instructions and constants (and parentheses, as long as they are balanced) is a valid Push program.

Many of Push's most powerful features derive from the provision of a data type (and a corresponding stack) for Push code. A rich library of code-manipulating instructions (based on Lisp's list-manipulation functions) allows programs to transform their own code and to execute the results of program transformations. Among other things, this can be used to implement subroutines, macros, and recursive control structures; an evolutionary computation system that evolves Push programs can thereby evolve modular programs without pre-specification of the number of modules and without the introduction of syntactic constraints. Explicit code manipulation can also be used to allow programs to produce their own offspring, rather than having offspring produced by hand-crafted genetic operators; the Pushpop system described below makes use of this capability.

2.2 THE PUSHGP GENETIC PROGRAMMING SYSTEM

PushGP is intended to be a fairly standard genetic programming system (see, e.g., Banzhaf et al., 1998) with the exception that it manipulates and produces programs expressed in the Push programming language. Figure 1 shows a flowchart for such systems. Details of the PushGP system, including the random code generation algorithm and descriptions of the built-in genetic operators, can be found in (Spector and Robinson, 2002).

Because PushGP manipulates and produces Push programs it can, without modification of the basic algorithm, produce programs that make use of multiple data types and programs containing subroutines and other possibly novel control structures. Nonetheless, many techniques that have been developed for more standard genetic programming systems can be applied to PushGP as well; for example, a recent study showed

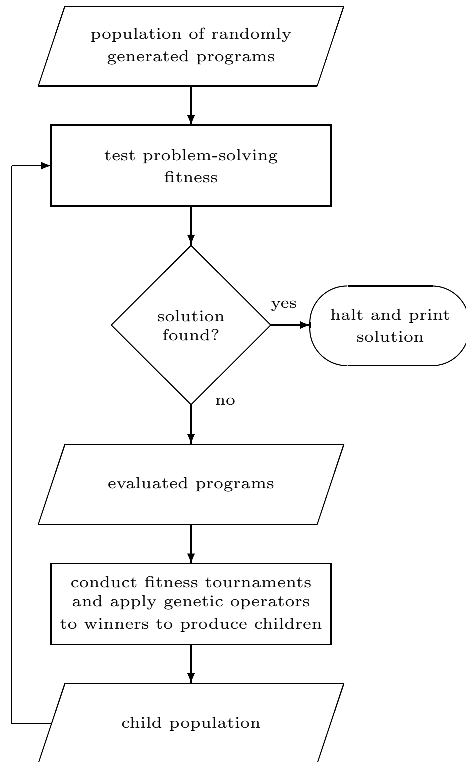

Figure 1: Flowchart of a standard genetic programming system with tournament selection; PushGP is a system of this type that manipulates and produces programs expressed in the Push programming language.

how "size fair" genetic operators, originally developed for tree-based genetic programming, can effectively control program bloat in PushGP (Crawford-Marks and Spector, 2002).

2.3 THE PUSHPOP AUTOCONSTRUCTIVE EVOLUTION SYSTEM

In (Spector and Robinson, 2002) we defined an *autoconstructive evolution system* to be any evolutionary computation system that adaptively constructs its own mechanisms of reproduction and diversification as it runs. Note that autoconstructive evolution systems, by constructing their own mechanisms of reproduction and diversification, thereby evolve their own evolutionary algorithms.

While a large body of previous work has explored "self-adaptive" evolutionary computation (Angeline, 1995; Angeline, 1996; Bäck, 1992; Stephens et al., 1998; Hart, 2000), autoconstructive evolution represents a more radical form of self-adaptation. In most previous work the algorithms for reproduction and diversification have been essentially fixed, with only the numerical parameters (such as mutation rates) subject to adaptation. By contrast, in an autoconstructive evolution system the individuals in the population are entirely responsible for constructing their own offspring.[1]

Pushpop, the name of which is derived from *Push* program *pop*ulation, is an autoconstructive evolution system that evolves Push programs. Pushpop inherits the skeleton of its algorithm from traditional genetic programming—it is a generation-based algorithm in which individuals are executed sequentially and in which the higher-performance individuals are allowed to contribute more children to the following generation. The basic algorithm of Pushpop is illustrated in the flowchart of Figure 2.

In contrast to the standard genetic programming algorithm of Figure 1, in Pushpop we do not normally terminate when a solution to the fitness-conferring problem is found, as we are often interested in observing the evolutionary dynamics even after such an event. More critically, evaluation of an organism's fitness in Pushpop produces not only a numerical measure of the organism's problem-solving ability but also a collection of potential children. These children are added to the following generation on the basis of fitness tournaments between their mothers—they themselves have not yet been tested for fitness, so their own fitness values cannot be used for this purpose. If there are not sufficient children to populate the next generation (in which case we say that the population is not yet "reproductively competent") then new organisms are randomly generated to pad the population. Pushpop programs can use their own code, randomly generated code, and the code of other programs in the population to construct their offspring. Details on these and other aspects of Pushpop can be found in (Spector and Robinson, 2002).

[1]Several "artificial life" systems in the literature (e.g. Tierra (Ray, 1991), Avida (Adami and Brown, 1995), and Amoeba (Pargellis, 1996)) may be considered autoconstructive in the sense used here, though the use of externally imposed mutation mechanisms in these systems may interfere with the evolution of diversifying reproduction. See (Spector and Robinson, 2002) for more discussion of this issue.

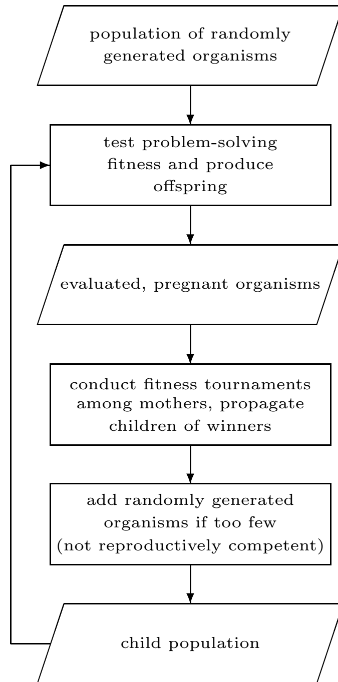

Figure 2: Flowchart of Pushpop, an autoconstructive evolution system that evolves Push programs.

3 EXPERIMENTS

3.1 EVOLVED TRANSPORT NETWORK CONTROL AGENTS

In collaboration with groups at MIT and BBN[2] we have applied PushGP to the evolution of agents in a transport network control problem. Similar problems have previously been investigated using more standard genetic algorithm techniques (Montana and Czerwinski, 1996). In this problem there are two types of agents: vehicles and network control agents. Vehicles move along linear corridors through a 2D or 3D environment, subject to control at intersections by the network control agents. In our simulations we use simple, hard-coded vehicle agents and attempt to produce network control agents that minimize the wait times of the vehicles across the network. In our first set of experiments we determined that PushGP could indeed produce agents that utilized their local sensor inputs to improve vehicular traffic flow. In our second set of experiments we looked at the ways in which the choice of training environments (used in fitness testing during evolution) impacts the robustness of the evolved agents across new environments with different characteristics. In the following paragraphs we briefly summarize this second set of experiments.

[2]Principally Oliver Selfridge and Wallace Feurzeig, as part of the DARPA ABC (Agent-Based Computing) TASK (Taskable Agent Software Kit) program.

3.1.1 Simulation

The simplest possible transport network was used in order to maximize the speed of evolution (see Figure 3). The network consisted of four one-way transit corridors crossing at one location. We designate the flow directions as North/South/East/West for convenience only—there is nothing inherently rectilinear or two-dimensional about the underlying simulation. In this setup, control agent status can be "green" (meaning "go") for North/South (and red/stop for East/West), or the reverse. Therefore, an agent is only required to specify the time-green value for North/South corridors.

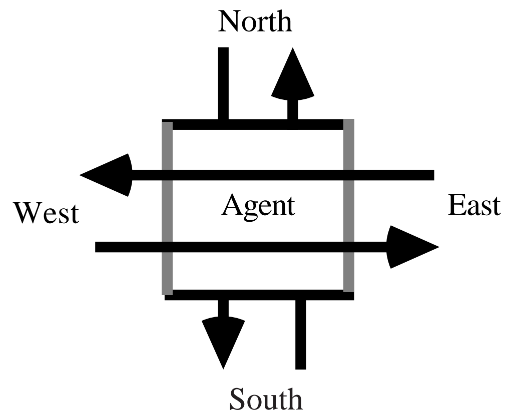

Figure 3: A simple transport network.

Each corridor is long enough to fit 20 vehicles on it at one time, and the control agents are located at the halfway point of each corridor. Until the first time the evolved agent is called a default value for time-green is used; this is 0.5 (green for half a control cycle). The agent's cycle length is a constant 20 time steps.

The simulation made available to the evolved agent the following data:

- Time green: amount of time the north/sound signal is in green mode. Expressed as float from 0.0 to 1.0. Higher means longer green.
- Average windowed wait: the amount of time spent

waiting (instead of moving forward) for the twenty most recent vehicles to enter each corridor, considering only those that have been in the grid for at least 5 time steps. This was gathered in terms of each vehicle, as well as an aggregate value for all the vehicles in the network. Expressed as a float between 0.0 and 1.0, where 1.0 means "yet to have moved," and 0.0 means "never had to stop."

- Maximum wait: the longest any vehicle has spent waiting for the entire history of the grid. Gathered in terms of each corridor, and an aggregate of all corridors. Expressed as a ratio of time spent waiting to total time spent in the simulation.

3.1.2 Runs

We performed two types of runs, "uniformly variable" runs and "discontinuous" runs, characterized by the nature of the environments used for fitness evaluation. The uniformly variable environments had stochastic flow patterns but no discontinuous transitions to new, radically different flow patterns during the course of the simulation. The discontinuous runs included more radical, discontinuous changes to flow patterns.

In the uniform runs agents were tested over 20 different fitness cases that consisted of different densities of traffic across the traffic grid. Fitness was measured in terms of average performance on each of the 20 different traffic densities. For each configuration of traffic density the simulation was run for 200 time steps. At every 20th time step, the evolved agent was evaluated to determine what new time-green value to use for the next 20 time steps. If the agent returned a value that was not between 0 and 1 then the pervious time-green value was used instead.

In the discontinuous runs the same 20 sets of traffic densities were used. The critical difference was that after running the simulation for 100 time steps the densities were changed to those of the next fitness case, without clearing the current vehicles from the grid. Then the simulation would continue for another 100 time steps before fitness would be assessed. In this way, every density configuration was used in evolution for the same total number of time steps, although it was spread across two different fitness tests.

7 runs, each of 80 generations and population size 2000, were completed for the uniform conditions and the discontinuous conditions. The best performing individuals from each run were subjected to a large number of additional tests in a wide range of environments to assess their performance and robustness.

3.1.3 Results

Figure 4 shows that the evolved programs, independent of the conditions of their evolution, fall into the same range of performance on 500 new uniformly variable trials. This cannot be said of discontinuous trials; most programs evolved in discontinuous conditions do better in discontinuous environments than do programs that evolved in uniform conditions. This supports our initial assumption that evolution in discontinuous environments would produce more robust agents. Nonetheless, the best performing program from the uniform trials performs as well as the best performing program from the discontinuous trials when retested in 500 new discontinuous environments.

One somewhat counterintuitive finding that emerged from detailed analysis of the evolved programs was that the programs evolved in uniformly variable environments were more immediately reactive to changes in their environments than were those evolved discontinuous environments. The programs evolved in the different environments appeared to adopt different control strategies, and those evolved in the discontinuous environments appeared to trade reactivity for longer-term robustness. More analysis will be required to understand how features of the Push language support such trade-offs and how they can be controlled.

A general lesson from this experiment is that considerable care must taken in selecting training environments for evolving agents; while too little variability may result in over-fitting to the training environments, too much variability may influence evolutionary dynamics in unpredictable ways.

3.2 MODULARITY AND ON-LINE EVOLUTION IN DYNAMIC ENVIRONMENTS

Terry Van Belle and David Ackley have recently conducted experiments using standard, tree-based genetic programming on a problem that changes over the course of evolution (Van Belle and Ackley, 2002). In particular they looked at symbolic regression of the formula $A*sin(A*x)$ where the constant A is randomly changed every epoch (5 generations). They found that ordinary genetic programming is completely stymied by the changes to A, with the result that significant adaptation never occurs. They also found, however, that genetic programming with an architecture that includes an ADF (automatically defined function) automatically discovers that the ADF can be used to isolate the part of the program responsive to the stable parts of the environment from the part of the program

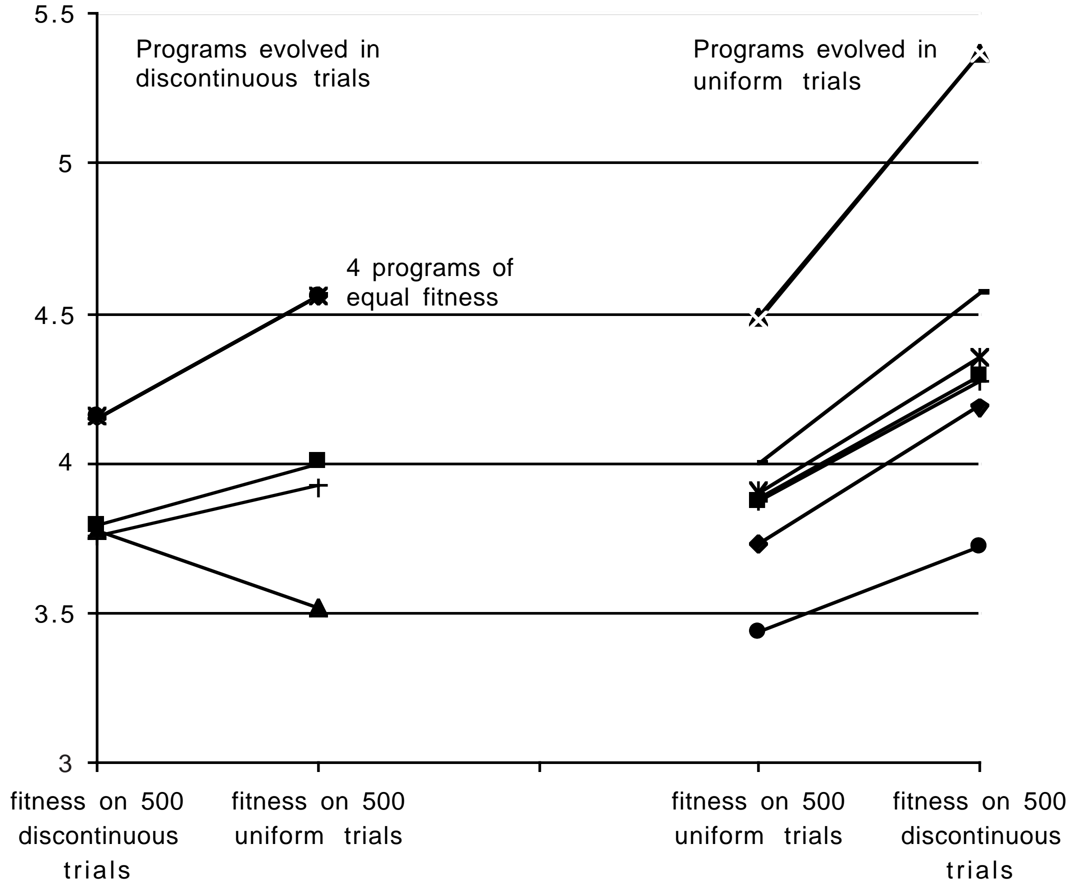

Figure 4: A graph comparing fitnesses obtained by evolving with discontinuous vs. uniformly variable trials. The lines on the left represent programs evolved in discontinuous trials; the lines on the right represent a separate set of programs evolved in uniformly variable trials. Lower fitness values are better.

responsive to the variable parts of the environment. As a result the system with an ADF was indeed able to adapt, and changes to A became less and less disruptive as runs progressed.

In PushGP the architecture of evolved programs (including the number of ADFs, if any) is not specified by the user but rather emerges as a product of natural selection. If PushGP is truly able to form modular programs automatically when it is helpful to do so, then we would predict that it would behave more like Van Belle and Ackley's runs with ADFs than like their runs without ADFs. To test this prediction we replicated the Van Belle and Ackley experiment using PushGP rather than standard tree-based genetic programming, using parameters similar to those used by Van Belle and Ackley.[3] The parameters that we used are listed in Table 1; see (Spector and Robinson, 2002) and documentation at `http://hampshire.edu/lspector/push.html` for the interpretation of these parameters.

As shown in Figure 5 we observed, as predicted, that PushGP was able to automatically induce modules and to thereby adapt over time so that changes to A became less disruptive. The graph shows average numbers of hits of best-of-generation programs at the start and end of each epoch, averaged over 144 runs. For standard genetic programming without ADFs these graphs would be essentially flat (though noisy); each change to A would destroy any prior increase in the number of hits. When an ADF is available Van Belle and Ackley showed that the hits increase as the epochs progress. Likewise, we observe here that with PushGP (and with no pre-specification of a modular architecture) the hits also increase as the epochs progress.

[3] Because of fundamental differences between the systems it was not possible to use completely identical parameters; for example, the instruction sets could not be identical because PushGP must have access to list-manipulation instructions in order to build and use modules.

Table 1: PushGP Parameters for Replication of the Van Belle and Ackley Experiments.

PARAMETER	VALUE
Population size	1000
Tournament size	5
Max generations	200
Fitness cases	50
Mutation %	45
Crossover %	45
Reproduction %	10
Mutation operators	standard, fair (0.25), perturb (50)
Crossover operators	standard, fair (0.25), uniform
Instruction set	ephemeral-random-integer, ephemeral-random-float, ephemeral-random-boolean, ephemeral-random-symbol, convert, =, rep, swap, pop, dup, max, min, >, <, /, ∗, −, +, pulldup, pull, exp, log, cos, sin, not, or, and, nth, list, cons, cdr, car, quote, map, if, do*, do, integer, float, boolean, type, code

PushGP does not appear to perform quite as well as standard genetic programming with an ADF here, but this is to be expected; PushGP must *discover* the modularization that is given to the standard system, and because of its larger instruction set (including list-manipulation instructions, etc.) it is searching a larger space of programs.

The general lesson from this experiment is that the capability for automatically creating new modules and otherwise altering program architectures can be useful for the on-line evolution of agents in dynamic environments. The experiment demonstrates that PushGP provides such a mechanism.

4 CONCLUSIONS

The Push programming language facilitates the evolution of agent programs using multiple data types, modules, and potentially novel control structures. Using PushGP, a genetic programming system that evolves Push programs, we briefly explored trade-offs in agent performance that result from greater or lesser dynamism in training environments. We also confirmed a result of Van Belle and Ackley demonstrating that the availability of modules (or module construction mechanisms) can allow an evolving agent to better adapt to partially stochastic environments.

Acknowledgments

This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30502-00-2-0611. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government. This research was also made possible by generous funding from Hampshire College to the Institute for Computational Intelligence at Hampshire College.

References

C. Adami and C. T. Brown (1995). Evolutionary Learning in the 2D Artificial Life System 'Avida', *Artificial Life IV*, MIT Press, 377–381.

P. J. Angeline (1995). Adaptive and Self-Adaptive Evolutionary Computations, in *Computational Intelligence: A Dynamic Systems Perspective*, IEEE Press, 152–163.

P. J. Angeline (1996). Two Self-Adaptive Crossover Operators for Genetic Programming, in P. J. Angeline and K. E. Kinnear, Jr., (eds.), *Advances in Genetic Programming 2*, MIT Press, 89–110.

T. Bäck (1992). Self-Adaptation in Genetic Algorithms, in *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, MIT Press, 263–271.

W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone (1998). *Genetic Programming: An Introduction*, Academic Press/Morgan Kaufmann.

W. S. Bruce (1997). The Lawnmower Problem Revisited: Stack-Based Genetic Programming and Automatically Defined Functions, in J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, and R.L. Riolo, (eds.), *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Morgan Kaufmann, 52–57.

R. Crawford-Marks and L. Spector (2002). Size Con-

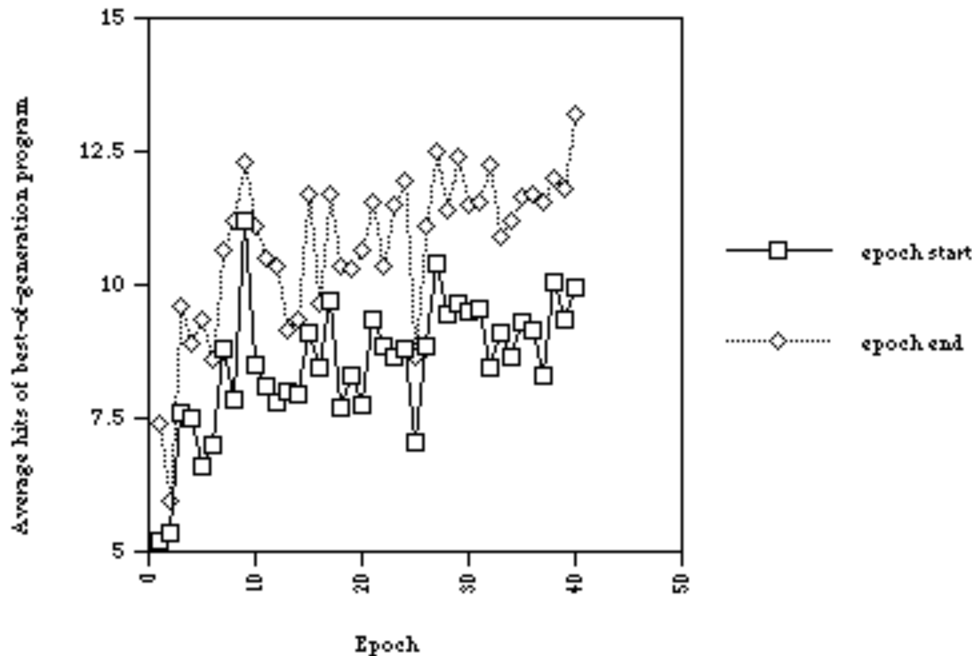

Figure 5: Data from runs of PushGP on the Van Belle and Ackley dynamic symbolic regression problem ($A * sin(A * x)$, with A changed randomly every epoch). The graph shows average hits of the best of generation program at the beginning and end of each epoch, averaged over 144 runs. An epoch is 5 generations.

trol via Size Fair Genetic Operators in the PushGP Genetic Programming System. In W. B. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002*, San Francisco, CA: Morgan Kaufmann.

W. E. Hart (2000). A Convergence Analysis of Unconstrained and Bound Constrained Evolutionary Pattern Search, *Evolutionary Computation* **9**(1), 1–23.

D. J. Montana (1995). Strongly Typed Genetic Programming. *Evolutionary Computation* **3**(2), 199–230.

D. J. Montana and S. Czerwinski (1996). Evolving Control Laws for a Network of Traffic Signals In *Genetic Programming 1996: Proceedings of the First Annual Conference*, MIT Press, 333–338.

A. N. Pargellis (1996). The spontaneous generation of digital life, *Physica D.* **91**, 86–96.

T. S. Ray (1991). Is it Alive or is it GA?, in *Proceedings of the Fourth International Conference on Genetic Algorithms,* Morgan Kaufmann, 527–534.

W. P. Salman, O. Tisserand, and B. Toulot (1984). *FORTH*, Springer-Verlag.

L. Spector (2001). Autoconstructive Evolution: Push, PushGP, and Pushpop. In L. Spector, E. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, 137–146. San Francisco, CA: Morgan Kaufmann Publishers.

L. Spector and A. Robinson (2002). Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* **3**(1)7–40.

C. R. Stephens, I. G. Olmedo, J. M. Vargas, and H. Waelbroeck (1998). Self-Adaptation in Evolving Systems, *Artificial Life* **4**, 183–201.

T. Van Belle and D. H. Ackley (2002). Code Factoring and the Evolution of Evolvability. In W. B. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002*, San Francisco, CA: Morgan Kaufmann.