Methods for Evolving Robust Distributed Robot Control Software: Coevolutionary and Single Population Techniques

Brad Dolin, Forrest H Bennett III, and Eleanor G. Rieffel
FX Palo Alto Laboratory, Inc.
3400 Hillview Avenue, Bldg. 4
Palo Alto, CA 94304
dolin@cs.stanford.edu, bennett@pal.xerox.com, rieffel@pal.xerox.com

Abstract

Previous work on evolving distributed control software for modular robots has resulted in solutions that do not generalize well to unseen test cases. In this work, we seek general solutions to an entire space of test cases. Each test case is a specific world configuration with a passage through which the modular robot must move. The space of test cases is extremely large, so a given training set can only be a sparse sample of this space. We look at several approaches for dealing with the problem of determining an effective training set: using a fixed set throughout a run, sampling randomly at each generation, and using coevolutionary approaches to evolve a population of test worlds. For this problem, random sampling outperformed the fixed sampling technique and did at least as well as the coevolutionary techniques we considered.

1. Introduction

Previous work on evolving distributed control software for modular robots has resulted in solutions that do not generalize well to unseen test cases [1]. In this work, we seek general solutions to an entire space of test cases. Each test case is a specific world configuration with a passage through which the modular robot must move. The space of test cases is extremely large, so a given training set can only be a sparse sample of this space. We look at several approaches for dealing with the problem of determining an effective training set: using a fixed set throughout a run, sampling randomly at each generation, and using coevolutionary approaches to evolve a population of test worlds. For this problem, random sampling outperformed the fixed sampling technique and did at least as well as the coevolutionary techniques we considered.

An effective sampling of training cases must meet two criteria. First, it must represent the space of all training cases well enough that the evolved solutions will be able to solve unseen cases. Second, it must be neither too hard nor too easy, as neither extreme provides a useful fitness signal to the evolving population of candidate solutions.

In the coevolutionary approaches there is an evolving population of robot control programs, and another population of evolving training cases. The motivation for this approach is twofold. First, the population of training cases may be able to adapt so as to avoid wasting fitness evaluations on cases that are too hard or too easy, and instead concentrate at each point during the learning process on training cases that provide as much information about the evolving candidate solutions as possible. Second, the population of training cases may be able to evolve to contain representative coverage of the total training space. This is because members of the training case population are rewarded for detecting and exploiting otherwise unnoticed strengths and weaknesses in the population of solutions. That is, consider any part of the training case space which is both not currently represented in the training case population, and solvable by only a few members of the solution population: such parts of the test space contain valuable training cases.

In the random sampling approaches, the set of training cases used are not being adapted to the population of robot control programs. These approaches were investigated as control experiments to determine whether the coevolutionary approaches were providing anything useful beyond random sampling.

2. Related work

The idea of evolving a maximally informative population of test cases has been explored in several domains. Hillis [9] and Juillé [10] coevolved sorting networks and test cases. Rosin [16] coevolved test cases for drugs and drug-delivery controllers. Sipper [18], Juillé and Pollack [11], and Werfel et. al. [21] coevolved cellular automata rules and initial configurations for the majority classification task. Haith et al. [7] devised a simple controller task, and coevolved initial conditions for the task. The work compares performance with coevolution against hand-tuned fitness scheduling.

A limited amount of work has been done on developing decentralized control software for modular robots [3, 4, 22]. Bennett et. al. describe the use of single-population genetic programming to develop control software for various simple modular robot tasks

[1], as well as a more involved task [2]. No work has been done on using coevolution to evolve distributed control software for modular robots.

3. Self-reconfigurable modular robotics

A self-reconfigurable modular robot is distinct from traditional robots in that it is composed of many simple, identical modules. Each module has its own power, computation, memory, motors, and sensors. The modules can attach to and detach from each other. Individual modules on their own can do little, but the robot, using the capabilities of the individual modules, can reconfigure itself to perform different tasks as needed.

Advantages of self-reconfigurable modular robots include physical adaptability to varying tasks and environments, robustness since identical modules can replace each other in the event of failure, and economies of scale in the manufacturing process. Physical implementations of modular robots include CMU's I-cubes [20], Dartmouth's Molecular Robots [12] and Crystalline Robots [17], MSU's Reconfigurable Adaptable Micro-Robot [19], Johns Hopkins University's Metamorphic Robots [14], as well as Xerox PARC's PolyBot [23], and Proteo robots [3].

Decentralized control takes advantage of the computational power of the individual modules and requires less communication bandwidth. All modules run the same program, but behave differently depending on individual sensor values, internal state, and messages received from nearby modules. The challenge is to design software that acts at the local level but achieves useful global behavior.

For our experiments, we wrote a simulator for the self-reconfigurable modular robots designed and built by Pamecha, et al. at Johns Hopkins University [14]. The square modules move in a two dimensional plane by sliding against each other. A module cannot move at all unless it is connected to another module that it can push or pull against. Each module occupies one grid unit in the simulated world, and all moves are in whole grid units. We add sensing, communication, and processing capabilities that were not implemented in the modules built by Pamecha, et al.

3.1. State

The state of a module includes its location, the most recent messages received from adjacent modules in each of eight directions, the values of its sensors, the values in its four memory locations, whether or not the module is broken, and its facing direction.

3.2. Directions

Directions are encoded as real values in [0.0, 1.0), where direction 0.0 is the robot module's positive x axis, and direction values increase going around counter clockwise. The direction values used by each robot module are local to that module's own frame of reference. The direction that a robot module is facing in the world is always direction 0.0 in the robot module's frame of reference. Direction 0.25 is 90 degrees to the left of where it is facing in the world, etc.

3.3. Sensors

Each module has eight sensors for detecting other robot modules, and eight sensors for detecting walls and obstacles. The eight directions for sensors are: 0.0 = east, 0.125 = northeast, 0.25 = north, 0.375 = northwest, 0.5= west, 0.625 = southwest, 0.75 = south, and 0.875 = southeast. A module's sensor readings are in units of intensity in [0.0, 1.0], where the intensity is the inverse of the distance to the thing sensed; zero means that the thing was not sensed at all, and one means that the thing was sensed in the immediately adjacent grid location.

3.4. Movement

Robot modules are able to move only in the four directions: east, north, west, and south. A robot module cannot move by itself; it can only move by sliding against an adjacent robot module.

To slide, a robot module can push itself against another robot module that is adjacent to it at 90 degrees from the direction of motion as demonstrated in Figure 1. Similarly, a robot module can pull against another robot module that is diagonally adjacent to it in the direction of motion. The pulling style of move is demonstrated in Figure 1 by considering step 2 as the initial configuration, and considering step 1 as the result of the move.

Robot modules can initiate two different types of moves to reconfigure the robot. A "single" move is the movement of a single robot module, and it succeeds if and only if that robot module is moving into an empty grid location and there is a module to push or pull against. A "line" move is the movement of an entire line of robot modules, and it succeeds if and only if the front-most module in the line is moving into an empty grid location, and there is a module to push or pull against. A line move can be initiated by any robot module in a line of robot modules.

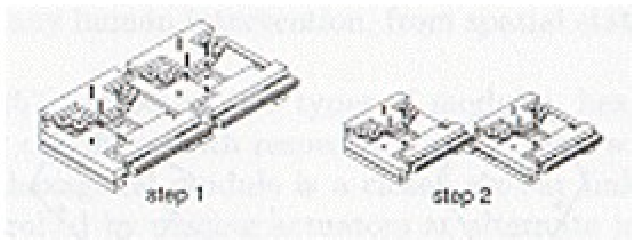

Figure 1. MoveSingle operation
This diagram depicts the basic MoveSingle operation for the self-reconfigurable robot modules used for the experiments in this paper. Before the MoveSingle, the two modules share a connected edge (step 1). After the MoveSingle the two modules are still connected at a shared corner (step 2). Reprinted by permission © ASME.

4. Robot control programs

Each robot control program is a lisp S-expression built from the following functions and terminals. The control program is run in each of the four modules that make up the robot for the experiments presented here.

4.1. Function set

- `(MoveLine direction)` causes an entire line of robot modules to move in the relative direction `direction` if they can. The line of modules to move is the connected line of modules collinear with the current module in the relative direction `direction` of the current module. The line of modules can move if and only if the front-most module in the line is not blocked by an immovable wall or obstacle and there is another module adjacent to the line that it can push or pull against. This function returns true if the line of modules is able to move, and false otherwise.
- `(MoveSingle direction)` causes the current robot module to move in the relative direction `direction` if it can move. The module can move if and only if it is not blocked by any other object, and there is another adjacent robot module that it can push or pull against. This function returns true if the line of modules is able to move, and false otherwise.
- `(ReadMessage direction)` reads the real-valued message from the adjacent module (if any) that is location at the relative direction `direction` to the receiving module. The direction is interpreted mod 1, and then rounded to the nearest eighth to indicate one of the eight adjacent grid locations. This function returns the value of the message read.
- `(ReadSensorSelf direction)` reads the intensity value of the self sensor. Intensity is the inverse of the distance to the closest robot module at the relative direction `direction`. The intensity is zero if there is no robot module in that direction, and the intensity is 1 if there is a module in the adjacent square. This function returns the intensity value.
- `(ReadSensorWall direction)` reads the intensity value of the wall sensor. Intensity is the inverse of the distance to the wall or obstacle in the relative direction `direction`. The intensity is zero if there is no wall or obstacle in that direction, and the intensity is 1 if there is a wall block in the adjacent square. This function returns the intensity value.
- `(Rotate direction)` rotates the robot module by the amount direction. This does not physically move the module, it just resets the internal state of the robot module's internal facing direction. This function returns the value of direction.
- `(SendMessage message direction)` sends the real-valued message `message` from the sending module to an adjacent module (if any) in the direction `direction`, relative to the frame of reference of the sending module. The direction is interpreted mod 1, and then rounded to the nearest eighth to indicate one of the eight adjacent grid locations. This function returns the value of the message sent.
- `(GetRegisterEntry index)` gets the current value of the robot module's memory numbered `index`. This function returns the value of the memory.
- `(SetRegisterEntry index value)` sets the value of the robot module's memory numbered `index` to `value`. This function returns value.
- `(If test arg1 arg2)`, and the remaining functions here, with the usual definitions [13].
- `(And test arg1 arg2)`
- `(Modulus arg1 arg2)`
- `(ProtectedDiv arg1 arg2)`
- `(LessThan arg1 arg2)`

4.2. Terminal set

- `(LessThan arg1 arg2)`
- `(GetTurn)` returns the current value of the `turn` variable for the simulation. The `turn` variable is set to zero at the beginning of the simulation and is incremented each time all of the robot's modules are executed.
- Six constant-valued terminals 0.0, 0.25, 0.5, 0.75, 1.0, and –1.0 are given. Program trees can use arithmetic to create other numerical values.

5. Test worlds

The simulation worlds for the modular robot are rectangular, and bordered on all sides by impassable walls. The modular robot is composed of four modules, and its initial configuration is a square on the bottom of the world. The initial facing direction of each module is east.

Each world optionally contains a horizontal wall, with a passage which may be straight, crooked, or crooked with a kink (Figure 2). Every such world can be completely specified by the ten parameters in Table 1.

Each test world is represented as a linear GA of length 10; each locus specifies a raw value, between 0 and 500, for the world parameters given above. Each raw value is then normalized to become the actual value for the relevant world parameter. We give the parameters and their possible domains in Table 1.

Table 1. Parameter domains

World Parameter	Min	Max
`a` := World width	*5*	*60*
`b` := World height:	*10*	*30*
`c` := Passage left X	*1*	*a - 3*
`d` := Passage bottom Y	*5*	*b - 3*
`e` := Passage width	*2*	*a - 1 - c*
`f` := Passage height	*0*	*b - 1 - d*
`g` := Width of crook	*0*	*e - 1*
`h` := Height of kink	*0*	*f - 2*
`i` := Handedness	*0*	*1*
`j` := Initial robot X	*1*	*a - 4*

The meanings for the variables in the right two columns are given in the leftmost column.

There is an additional domain restriction on the height of the kink. When the crooks extend across approximately half the passage, tall kinks would close off all paths from the bottom of the world to the top. In such situations, the kink height is reduced to one less than half the height of the passage, in order to maintain an open path.

This normalization scheme will only create worlds with a path from bottom to top of at least one grid unit wide. However, it should be noted that some worlds do not permit movement of the robot from bottom to top, because they may contain long, narrow pathways which cannot be traversed by the style of robot we experiment with here.

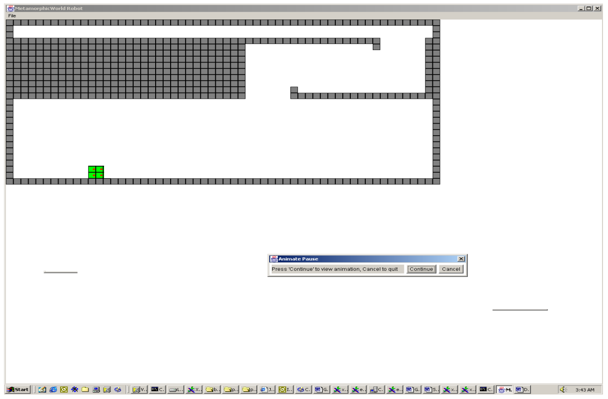
Animate Pause
Press 'Continue' to view animation, Cancel to quit

Figure 2. A typical world
The “crook” is the wall that extends horizontally into the passage, and the “kink” extends vertically into the passage. “Handedness” is 0 if the top crook is on the left, and 1 if it is on the right.

6. Control program evolution

6.1. Competition

A competition is said to occur whenever a control program is simulated in a test world, as described below. The outcome of the competition is a Boolean value, which indicates whether or not the modular robot was able to successfully make it through the passage to the top portion of the world. We certainly could have defined a real-valued competition outcome which gives credit for partial solutions. However, we chose to use an extremely coarse measure with an eye toward applying this technique on problem domains where assignment of partial credit is either impossible or unlikely to meaningfully guide evolution.

Specifically, the control program wins the competition (solves the problem) if and only if all of its modules make it to the two top rows in the world. The competition outcome is anti-symmetric, so that the test world loses exactly when the control program wins.

6.2. Simulation

A simulation of the robot’s actions in the world is run for a number of turns equal to twice the height of the world. In each turn, the evolved program tree is executed one time for each module, as shown in this pseudocode:

```
for (turn = 0; turn < maxTurns; turn++)
   for (module = 0; module < numModules; module+
+)
         programTree.eval(module);
```

While executing the program tree for a single module, the sensor readings, the primitives that access the internal state of the modules, and the movement primitives are all executed for that one module. The order of module execution remains fixed throughout the run. If the robot makes it to the top of the world, the simulation is terminated at this time step.

6.3. Competitive fitness sharing

The fitness measure is used to determine the relative performance of each individual, and is relevant during the evolutionary selection process. For the methods in this paper, higher fitness is better.

We use competitive fitness sharing [15] to determine the fitness of control programs in all experiments given here. With this technique, each opponent in the sample is seen as a resource to be shared among all those individuals that defeat it. That is, if *Nj* individuals defeat individual *j* from the sample, then each individual *i* who defeats *j* gets *1 / Nj* added to its fitness score.

Competitive fitness sharing is useful for maintaining phenotypic diversity, because it rewards individuals not only for defeating many opponents in the sample, but also for defeating opponents which few other individuals defeat.

6.4. Shared sampling

When we are using coevolutionary methods, sampling selects some subset of the current population for competition against each individual in the opponent population. We sample only some fraction of the population to minimize the number of costly simulations, but we use the competition results to choose a sample that is likely to provide meaningful gradient for evaluation of the opponent population.

We use the shared sampling technique [15] to choose the control program sample for all experiments given here. The algorithm is described as follows:

1. Calculate the "sample fitness" of each individual not in the current sample, *S*. The sample fitness of an individual *i* is simply the fitness it would receive under competitive fitness sharing, described above, computed over the population $S \cup \{i\}$.
2. Add the opponent with the highest sample fitness to the sample.
3. Loop to 1 until we have reached the desired sample size.

The algorithm attempts to choose a sample of adept control programs which defeat a diverse set of test worlds.

6.5. GP parameters

The internal node crossover rate is 79%; the leaf node crossover rate is 1%; the internal node mutation rate is 9%; the leaf node mutation rate is 10%; and the cloning rate is 1%. We use the generational breeding model with tournament selection, and a tournament size of 4. Elitism is used, which ensures that the most fit individual in each generation is cloned into the next generation. The method for creating program trees in generation 0 and in mutation operations is the "ramped half and half" method [13]. The maximum depth for program trees in generation 0 is 6. The maximum depth for program trees created by crossover and mutation is 10. The maximum depth of subtrees created by mutation is 4. We use strongly typed genetic programming [8]. The population size is 500, and we sample 5 individuals for opponent testing (discussed below).

7. Test world evolution

When the population of test worlds is also being evolved, we use the following coevolutionary algorithm:

1. Construct initial random populations
    a. Control programs
    b. Test worlds
2. Choose a *random* sample of test worlds
3. Compete each control program against the sample of test worlds
4. Compute the fitness of each control program
5. Choose an informed sample of control programs, using shared sampling
6. Evolve the next generation of control programs using GP operators
7. Compete each test world against the sample of control programs
8. Compute the fitness of each test world
9. Choose an informed sample of test worlds, using one of the methods described below
10. Add the three highest fitness test worlds to the hall of fame
11. Evolve the next generation of test worlds using GA operators
12. Loop to 3

Each loop through the algorithm creates a new generation of individuals in each population.

7.1. Fitness measures

The test worlds possess an inherent competitive advantage: only a small fraction of the randomly created control programs will be able to defeat any test worlds at all. Thus, although we attempt to evolve a challenging and diverse set of test worlds, we also need to ensure that this set of problems is *tractable* by the learners, providing a meaningful gradient for evolution. To this end, we experiment with two methods for measuring fitness:

Tractable Competitive Fitness Sharing. The fitness of test worlds is computed using competitive fitness sharing, with the caveat that worlds which are not defeated by any control programs in the sample receive worst fitness. Since such intractable individuals may later prove useful, once the population of learners has become more adept, we do not eliminate them completely from the population. They have slim probability of surviving into the next generation, which will occur only if a set of 4 worst fitness individuals is chosen for a tournament. Additionally, with probability *1 / population_size*, an intractable individual is added to the hall of fame, discussed below.

Distinction Fitness. One might argue that we should measure the fitness of each test world (teacher) according to its ability to distinguish between performance abilities of control programs (learners). To this end, we use a version of one of the fitness measures given in Ficici and Pollack for the population of teachers [6]. To compute the distinction fitness for an individual i, we first determine all pairs of learners which this teacher is able to distinguish, i.e., defeat one but not the other. For each such pair p, we add $1 / N_p$ to i's distinction fitness, where N_p is a discount factor giving the total number of teachers in the population which distinguish the pair p.

7.2. Sampling methods

In addition to shared sampling, we experimented with several additional techniques:

Random Sampling. Individuals are chosen at random from the current population.

Uniform Sampling. For uniform sampling, we first rank the entire population of opponents according to fitness. For a sample of size S, we then divide the population into S equally sized bins, where each bin contains opponents with similar fitness. We then randomly choose one member for the sample from each bin.

While uniform sampling may appear quite similar to random sampling, since we are picking individuals from the population with uniform probability, there is a subtle difference. With small sample sizes, random sampling will, from time to time, result in an anomalous sample – one that is intractable, too easy, or genetically uniform. Uniform sampling attempts to avoid this difficulty by ensuring that individuals come from different segments of the population

Distinction sampling. This technique is relevant only when distinction fitness is used. The method is analogous to shared sampling, except that sample fitness is calculated based on the opponent pairs that each individual is able to distinguish (instead of the opponents defeated by each individual). The goal is to choose a sample of trainers which distinguishes between a large and diverse set of control programs.

7.3. Hall of fame

To help avoid the problem of overspecialization to the current generation of opponents, we utilize the hall of fame technique [16] to store best-of-generation individuals from the population of test worlds. The three best individuals from each generation are added to the hall of fame; if tractable competitive fitness sharing is being used, a small number of undefeatable individuals will be added as well, as discussed above. Then, each control program's fitness is determined with respect to its performance against the regular sample of test worlds union some random sample from the hall of fame.

7.4. GA parameters

Elitism is used. The population size is 100, from which we sample 6 individuals for opponent testing. We sample 3 individuals from the hall of fame. This gives a total sample size of 9 worlds for control program testing. We use the generational breeding model with tournament selection, and a tournament size of 2. There is no crossover. The only genetic operator is cloning with Gaussian mutation at a rate of 50% at each locus. Mutation replaces the value at each locus with a random value normally distributed about the old, with standard deviation of *0.1 * max_allele_value = 50*. If a mutation would bring the allele value out of bounds, it takes on the extreme legal value.

8. Experimental setups

We compare the performance of two setups in which only the population of control programs is evolving (R1 and R2), and four setups in which both the population of control programs and the population of fitness cases are evolving (C1 – C4). Note that for all setups, we measure the fitness of individuals in the population of control programs according to competitive fitness sharing, and, for the coevolutionary techniques, we choose its sample via shared sampling. We vary the fitness measure and sampling technique for the population of trainers.

R1. Random static: A randomly generated, static sample of 10 worlds is used to evaluate control programs.

R2. Random dynamic: A sample of 10 worlds, chosen at random each generation from a randomly generated population of 100 worlds, is used to evaluate control programs.

C1. Distinction fitness with distinction sampling for the test world population.

C2. Distinction fitness with random sampling for the test world population.

C3. Competitive fitness sharing with uniform sampling for the test world population.

C4. Competitive fitness sharing with shared sampling for the test world population.

9. Results

Results are averaged over 8 independent runs. We measure the performance at each generation by evaluating the control program which defeats the largest fraction of the current test world sample on an out-of-sample test set. The out-of-sample test set contains 500 randomly generated test worlds, held fixed over all runs and over all generations. Figures 3 – 7 show the fraction of this test set (vertical axis) solved by the individual with the most wins at each generation (horizontal axis).

Note that, at each generation, the number of program-world competitions, 5000, is the same for all methods. At each generation, random static and random dynamic compete 500 control programs against 10 worlds each. The coevolutionary methods compete 500 control programs against 9 worlds each when evaluating control programs, then 100 worlds against 5 programs each when evaluating worlds.

Experimental results fail to distinguish between the best random method and the best coevolutionary method; compare Figure 3 to Figures 4 - 7. Interestingly, we do find that the random dynamic method significantly outperforms the random static method; see Figure 3.

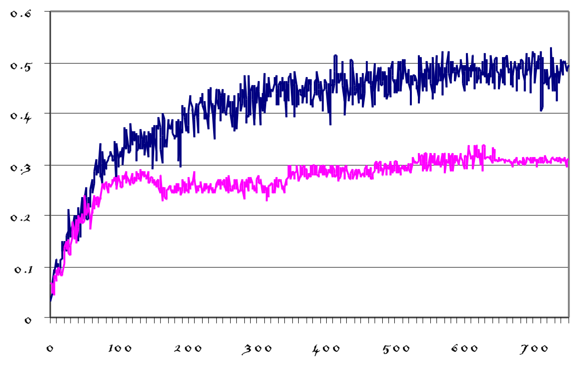

Figure 3. Random dynamic, top vs. Random static, bottom

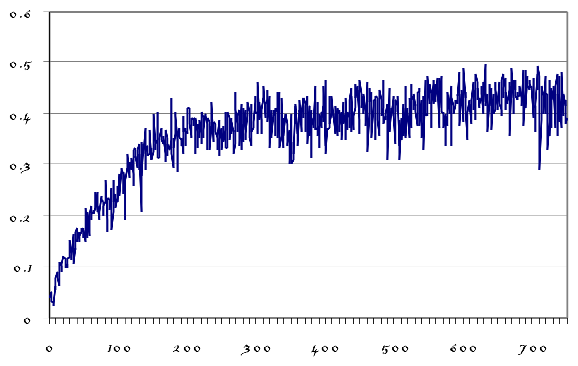

Figure 4. Distinction fitness with distinction sampling

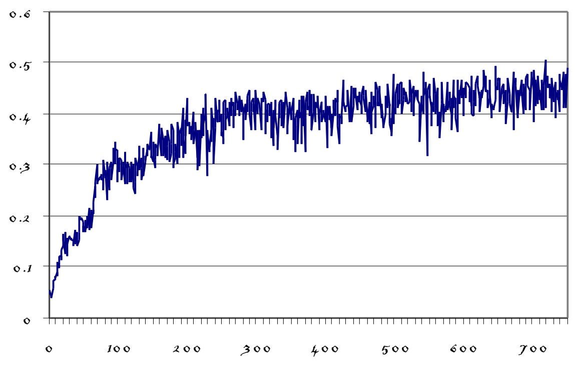

Figure 5. Distinction fitness with random sampling

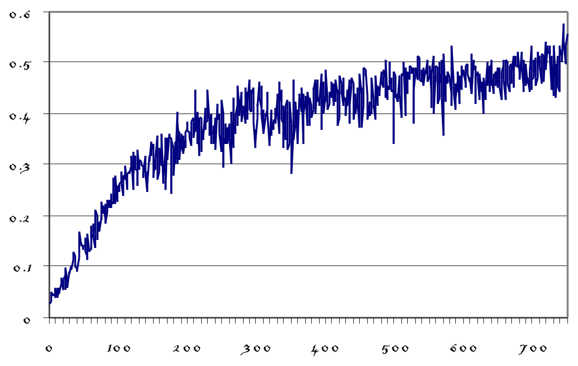

Figure 6. Shared fitness with uniform sampling

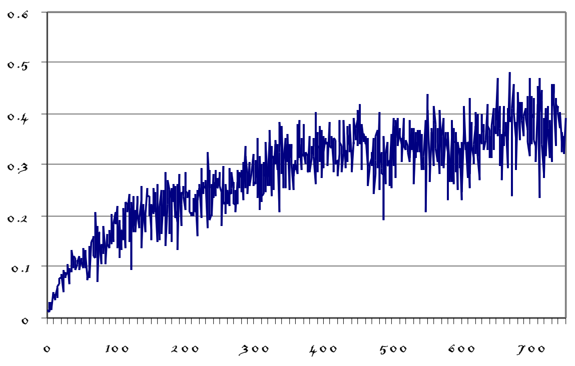

Figure 7. Shared fitness with shared sampling

10. Conclusions and future work

We compared several techniques for evolving modular robot control software which solves a wide variety of test worlds. We found that evaluation with a changing set of test worlds was more effective than evaluation with a fixed set. The hope was that sacrificing one member of the test world sample, and using the extra simulations to instead evolve and choose a more informed sample, would lead to the evolution of better control programs.

However, for this problem, the best random method was as effective as the best coevolutionary method in evolving control programs which exhibit robustness to out-of-sample test worlds. We plan to investigate means of improving the performance of coevolutionary methods for this problem.

Acknowledgments

The authors gratefully acknowledge Nils Nilsson and John Koza for fruitful conversations about issues in coevolution. Many thanks to Tad Hogg and Arancha Casal for teaching us about modular robots and suggesting problem areas for us to work on, Wolfgang Polak for help in setting up the computing cluster and for conversations about modular robots, and Adil Qureshi for his Gpsys 1.1 Java genetic programming source code on which our system is based.

References

[1] Forrest H Bennett III, Eleanor G. Rieffel, "Design of Decentralized Controllers for Self-Reconfigurable Modular Robots Using Genetic Programming", Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware, 2000.

[2] Forrest H Bennett III, Brad Dolin, Eleanor G Rieffel, "Programmable Smart Membranes: Using Genetic Programming to Evolve Scalable Distributed Controllers for a Novel Self-Reconfigurable Modular Robotic Application", Proceedings of the 4th European Conference (EuroGP), 2001.

[3] Hristo Bojinov, Arancha Casal, Tad Hogg, "Emergent Structures in Modular Self-reconfigurable Robots", IEEE Intl. Conf. on Robotics and Automation (ICRA), 2000.

[4] Eric Bonabeau, Marco Dorigo, Guy Theraulaz, Swarm Intelligence: from Natural to Artificial Systems, Oxford Univ. Press, 1999.

[5] Arancha Casal, Mark Yim, "Self-Reconfiguration Planning for a Class of Modular Robots", Proceedings of SPIE'99, Volume 3839, 1999.

[6] Sevan G. Ficici, Jordan B Pollack, "Pareto Optimality in Coevolutionary Learning", Computer Science Technical Report CS-01-216, 2001.

[7] Gary L. Haith, Silvano P. Colombano, Jason D. Lohn, Dimitris Stassinopoulos, "Coevolution for Problem Simplification", Proc. 1999 Genetic and Evolutionary Computation Conference, (GECCO-99), 1999.

[8] Thomas Haynes, Roger Wainwright, Sandip Sen, Dale Schoenfeld, "Strongly Typed genetic Programming in Evolving Cooperation Strategies", in Proceedings of the Sixth International Conference on Genetic Algorithms, 1995.

[9] Daniel W. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," Artificial Life II, Addison-Wesley, 1991.

[10] Hugue Juillé. "Incremental co-evolution of organisms: a new approach for optimization and discovery of strategies", Proceedings of the Third European Conference on Artificial Life, Springer-Verlag, 1995.

[11] Hugue Juillé and Jordan B. Pollack, "Coevolving the 'Ideal' Trainer: Application to the Discovery of Cellular Automata Rules", Proceedings of the Third Annual Genetic Programming Conference, 1998.

[12] K. Kotay, D. Rus, M. Vona, C. McGray, "The Self-reconfiguring robotic molecule: design and control algorithms". In Proceeding of the 1998 International Conference on Intelligent Robots and Systems, 1998.

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

[14] A. Pamecha, C. J. Chiang, D. Stein, G. S. Chirikjian, "Design and Implementation of Metamorphic Robots", In Proceedings 1996 ASME Design Engineering Technical Conference and Computers and Engineering Conference, 1996.

[15] Christopher D. Rosin and Richard K. Belew. "Methods for competitive co-evolution: Finding opponents worth beating", Proceedings of the Sixth International Conference on Genetic Algorithms, 1995.

[16] Christopher D. Rosin, Coevolutionary search among adversaries, Ph.D. thesis, University of California, 1997, San Diego.

[17] D. Rus, M. Vona, "Self-Reconfiguration Planning with Compressible Unit Modules", In 1999 IEEE Int. Conference on Robotics and Automation, 1999.

[18] Moshe Sipper. "Co-evolving non-uniform cellular automata to perform computations", Physica D, 1996.

[19] R. L. Tummala, R. Mukherjee, D. Aslam, N. Xi, S. Mahadevan, J. Weng, "Reconfigurable Adaptable Micro-

Robot", Proc. 1999 IEEE International Conference on Systems, Man, and Cybernetics, October 1999.

[20] C. Ünsal, H. Kiliççöte, P. K. Khosla, "A 3-D Modular Self-Reconfigurable Bipartite Robotic System: Implementation and Motion Planning", submitted to Autonomous Robots Journal, special issue on Modular Reconfigurable Robots, 2000.

[21] Justin Werfel, Melanie Mitchell, and James P. Crutchfield. "Resource sharing and coevolution in evolving cellular automata", Submitted to IEEE Trans. Evol. Comp., 1999.

[22] Mark Yim, John Lamping, Eric Mao, J. Geoffrey Chase, "Rhombic Dodecahedron Shape for Self-Assembling Robots", Xerox PARC SPL TechReport P9710777, 1997.

[23] Yim, M., Duff, D.G., Roufas, K.D., "PolyBot: a Modular Reconfigurable Robot", IEEE Intl. Conf. On Robotics and Automation (ICRA), 2000.