Multi-population competitive co-evolution of car racing controllers

Julian Togelius, Peter Burrow and Simon M. Lucas, *Senior Member, IEEE*

***Abstract*— Multi-population competitive co-evolution is explored as a way of developing controllers for a simple (but definitely not trivial) car racing game. The three main uses we see for this method are to evolve more complex general intelligence than would be possible with other methods, to compare different evolvable architectures for controllers, and to develop behaviourally diverse populations of agents for computer games. Nine-population co-evolution is compared with single-population co-evolution and standard evolution strategies, steady-state and generational versions of the algorithm are compared, and a number of different controller architectures are compared with each other.**

I. Introduction

Can we use competitive co-evolution to develop complex, high-performing artificial intelligence for computer game agents? Further, can we use it to develop *interesting* and *entertaining* game agent AI? The answer to the first question seems to be yes, at least for some games under some conditions - see e.g. recent experiments with board games [1] and space shooters [2]. But competitive co-evolution has its own set of problems, notably the cycling problem, which most likely need to be sorted out before we can make further progress. The second question seems hardly to have been addressed at all in the literature.

This paper concerns the evolution of game agent AI, more specifically car racing controllers, through multi-population competitive co-evolution. The originality lies mainly in the use of more than two populations in the competitive co-evolution, something which is severely understudied.

A. Evolutionary car racing

We have in a series of papers over the last two years investigated the application of evolutionary algorithms (and some other forms of machine learning) to car racing simulations of modest complexity. The basic experiments, reported in [3], showed that very good controllers based on neural networks can be evolved for a single car racing on a single track, using only progress on the track as the fitness function. In further experiments, we studied the incremental evolution of more general driving skills [4], and single-population co-evolution of car controllers for two-car races in [5]. Some other recent papers deal with modelling the driving style of human players and evolving tracks that are fun to drive [6], and modelling the dynamics of real toy cars so as to be able to transfer evolved controllers from simulation to reality [7].

The evolutionary car racing project has at least three different motivations. One is to develop a set of benchmarks for algorithms for learning control. Another is to develop ways for generating interesting racing game content, both car controllers and other sorts of content, such as racing tracks. But the original and most important motivation is to make progress on the grand goal of evolving complex general intelligence. We believe much can be learned from studying evolution of game playing agents, as computer games are in many respects ideal for evolving intelligence in.

Julian Togelius and Simon M. Lucas are with the Department of Computer Science, University of Essex, Colchester CO4 3SQ, United Kingdom. Peter Burrow is an independent researcher in Cambridge, United Kingdom. (emails: jtogel@essex.ac.uk, prb26@cantab.net, sml@essex.ac.uk).

B. Choosing a controller architecture

Although we have shown that we can evolve neural network-based controllers that successfully tackle different car racing problems, this is by no means the final word on how to construct such controllers. In theory, evolvable controllers could be represented in innumerable ways including genetic programming and nearest-neighbour classifiers. That feedforward multi-layer perceptrons (MLPs) are sufficient for the control task in question does not by any means imply that they are optimal. In fact, several objections can be raised.

Firstly, MLPs are stateless, and a controller based only on an MLP is reactive and thus unable to integrate information over time. Being able to do this might well give a competitive edge in certain car racing tasks, whenever the information available from the sensors at any one time is insufficient for a given action. An example could be estimating whether a competing car is accelerating or decelerating (based on instantaneous velocities) so as to know whether to overtake.

Another issue is modularity. In an MLP, every neuron in a layer is connected to every other. However, many studies have shown that having too many connections can actually make learning harder, whether evolution or some other learning algorithm is used [8]. The solution here is to divide the network into sub-networks, or otherwise modularise the controller.

And let us not forget something so simple as the size of the controller, and therefore the search space. Can larger controllers represent more complex strategies? Further, is there a tradeoff between the size of the controller and the learnability? There is no consensus on this topic, but note the concept of the *extradimensional bypass* which suggests that larger neural networks could learn faster by avoiding local minima [9].

C. Competitive co-evolution

Competitive co-evolution is when the fitness function of an individual is made dependent on other individuals, either in the same population, or in a different population altogether. The promise of co-evolution is that linking the fitness in this way will lead to some form of global improvement, as individuals compete against each other.

The idea is to encourage an evolutionary "arms race", where improvements in some individuals cause further improvements in other individuals, and vice-versa.

It quickly became clear however, that such co-evolutionary schemes can be prone to complex dynamics, which can thwart global progress towards higher fitness. There has also been a implicit assumption that an evolutionary arms race leads directly to an increase in complexity, though this is not always the case.

An example of a potential problem with co-evolution is that of "cycling" between different strategies. If an individual develops a strategy that affords it a higher fitness relative to other individuals, it will spread through the population, which will stabilise until a strategy arises that exploits a weakness in the previous one. There is then another rapid replacement of individuals. However, there is no guarantee that the new strategy is better than the one its predecessor replaced, as the dominance relation is intransitive. It is entirely possible for the population to cycle through the same set of possible strategies, each exploiting the weaknesses of the previous one, without any global increase in fitness.

Another problem that can occur in the case of more than one population is *disengagement*, a loss of selective gradient. This is when individuals in one population consistently beat individuals in another population, destroying the selection pressure on those individuals.

Several attempts have been made to address these problems, the most prominent of which was invented by Rosin and Belew: the "hall of fame" [10]. This technique has the individuals of the current generation compete not only against other current individuals, but also against a selection of good individuals from previous generations. Exactly how this should be done has been the subject of several studies, and it seems the proposed solution poses its own problems.

D. Steady-state versus generational selection

Standard co-evolution proceeds in generations. In each generation there is a period of evaluation, followed by population decimation and replacement.

Though this generational scheme is the standard approach to co-evolution in use today, it is not the only way that co-evolution can proceed. In nature, the process of replacement is usually less dramatic - populations usually remain stable and there is continuous replacement of individuals.

Miconi and Channon [11] introduce one method of performing steady-state co-evolution. The N-strikes-out algorithm they propose performs both evaluation and replacement asynchronously, on an individual basis. To the authors' knowledge, this is the first truly steady-state co-evolutionary algorithm.

This asynchronous updating means that the selection landscape changes gradually, in effect acting as a self maintaining archive of previous fit individuals, and avoiding the need for a hall of fame. The motivation is that this will discourage exploits of the current champion's weaknesses, as there is more likely to be other high fitness individuals which don't share that specific weakness.

E. Multi-population co-evolution

The vast majority of experiments in *competitive* co-evolution use one or two populations only (even though there are several examples of collaborative multi-population co-evolution). That this territory seems so uncharted is surprising, as there are reasons to believe that there could be many advantages to competitively co-evolving more than two populations. Janzen distinguished between *true* and or *diffuse* co-evolution [12] in the context of evolutionary biology (not computer science). The former is defined as evolutionary change in a specific trait of one population in response to change in a specific trait of another population; the latter is defined as *non-specific* evolutionary change in response to a *group* of traits.

Bullock argued that diffuse rather than true co-evolution would be desirable from an engineering standpoint, as diffuse co-evolution should lead to more robust solutions [13]. Multi-population co-evolution could be a way to achieve this. This was done by Hornby and Mirtich in a predator-prey simulation [14]. Our own take on this is to use multiple populations to try to force diffuse symmetric co-evolution, but also to use it to compare controller architectures. (In Hornby and Mirtich's work, only one controller architecture was used.) In the process, we compare two different selection strategies: steady-state and generational selection.

Our main hypothesis is that using many populations, on a problem where different strategies are possible, would automatically lead to diversification *between* (although not neccessarily within) the populations. This diversity could have several uses. First of all, it could help counteract cycling. If all individuals are tested against individuals of all other populations, and thus against a number of differing strategies, a narrow-focused strategy that only beats a particular other strategy would have little luck. The selection pressure would instead be on a good general strategy, that beats as many as possible of the other strategies. At the same time, all populations would probably not converge to the same strategy, because as soon as an empty niche in strategy space appears it would be most advantageous to fill that niche.

Apart from potentially helping us overcome the cycling problem, the diversity between the populations could be interesting in itself, especially from a computer game perspective. The challenging part of developing controllers for computer game agents is often not to make the agent play the game as well as possible, but to make the agent play in as interesting a manner as possible, thus heightening the satisfaction of the human player. To take car racing as a not very far fetched example, we might want to let the player compete against a starting field of diverse drivers, that each drive the track well but use different strategies to take turns and overtake other cars.

F. Scope of this paper

The main question we address in this paper is what is it possible to do with many-population co-evolution. Can

we evolve controllers that perform better than those evolved with solo-evolution, or one- or two-population co-evolution? Can we use many-population co-evolution to investigate the relative benefits of different controller architectures? When seeding a number of populations with the same architecture, can we evolve a behaviourally diverse set of controllers?

We are also interested in the relative performance of the steady-state and generational multi-population co-evolutionary algorithms. Particularly, we wonder whether the N-strikes selection mechanism manages to further alleviate the cycling problems.

And in addition to the issue of how to best compare a number of controller architectures, we are of course interested in the results of the comparison: which of the implemented controller architectures is best for the task given? For us, the underlying motivation is to be able to evolve complex general intelligence; studying the properties of particular controller architectures and evolutionary algorithms is a means to that end, rather than the other way around.

II. Methods

Below, we describe the agent, its environment, the fitness function, the various ways of representing the controllers that we implemented, and our co-evolutionary algorithms.

A. Car racing task

The racing game in which we evolve controllers, and the fitness function with which we score them, is the very same as was used for the Car Racing Competition at the IEEE Computational Intelligence and Games Symposium, organized by two of the authors and building on the experiences from earlier experiments in evolutionary car racing. More details about the competition can be gathered and its complete source code downloaded from http://julian.togelius.com/cig2007competition.

In this game, one or two cars compete to reach as many way points as possible within a set time. The basic fitness of a controller is calculated as the mean number of way points reached in five trials of 1000 time steps each. Way points are randomly positioned within a circular radius at the start of each trial; at any point two way points are visible to human or algorithmic players, but only one way point (the current way point) can be "taken" by a car at any occasion. As soon as the current way point is reached, the way point counter is incremented for that car, the other visible way point (the next way point) becomes the current, and a new next way point is generated at a random position.

The car control is intended to qualitatively mimic that of real radio-controlled toy cars, and so the cars have bang-bang control: back, back-right, back-left, neutral, neutral-left, neutral-right, forward, forward-left and forward-right are the available actions to take at any time step. The dynamics of the car are reasonably realistic, so acceleration and deceleration take time, and turning while traveling at high speed will cause considerable skidding. Turning on the spot is certainly not possible. As this version of the car simulation lacks walls, wall collisions are not implemented, but car-to-car collisions are possible and result in both vectorial and angular impetus.

Despite its apparent simplicity, this game has plenty of hidden depth. On one level it is just about driving straight for the current waypoint. Or, it would be, if it wasn't for the skidding at high speeds; an unsophisticated controller that doesn't slow down or reverse when the way point is within the turning radius for the current speed will end up orbiting the way point. And then there is the issue of passing the current way point at such an angle and speed that reaching the next way point is as easy as possible. Adding another car controlled by a competing controller to the game increases the complexity further, as the next way point will become the current whichever car reaches it first. So when the other car is likely to reach the current way point first, it will probably be a good idea to the head for the next way point directly. But being able to accurately predict whether this would be the case would require knowing not only the position, angle and speed of your competitor, but also its behaviour, and of course the dynamics of the cars. In fact, in some cases the best strategy might be to block your competitor by colliding with it, providing you can predict and leverage the outcome of the collision to your advantage...

B. Controller architectures

A number of evolvable controller architectures were implemented, for purposes of comparing speed and quality of learning. All controller architectures are based on one or two evolvable function approximators (in all cases except one these are neural networks), and in an all cases the main function approximator outputs two real numbers. These two numbers are interpreted as follows: if the first output is above 0.3, the driving force is set to forward, if below -0.3 the driving is to backward, and otherwise driving set to neutral. The second output decides whether to set the steering for current timestep to left, right or centre in the same way.

1) MLP controllers: The MLP controller is based on a standard multi-layer perceptron with 8 inputs, 6 hidden neurons, 2 outputs and *tanh* activation function. The inputs to the network are the speed of the car, the angle to the current way point, the distance to the current way point, the angle to the next way point, the distance to the next way point, the angle to the other vehicle, and the distance to the other vehicle (both the last values are set to 0 if there is no other vehicle present). Apart from these inputs, a bias input (always set to 1) is added to the all neural networks described in this paper. All angles are calculated as the difference between the orientation of the car and a straight line to the waypoint or competitor car in question.

At the start of an evolutionary run, all connection weights of all neural networks are set to zero. Mutation consists of adding random numbers drawn from a Gaussian distribution with standard deviation 0.1.

2) Recurrent controllers: Most of the controllers are based on simple recurrent neural networks, commonly known as Elman networks [15]. The recurrent neural networks are implemented as standard MLPs, with extra connections from

the hidden layer of the last time step to the hidden layer of the current time step.

Several controller architectures based on such networks are compared. The RMLPSmall, RMLP and RMLPBig controllers are all based on recurrent networks with exactly the same inputs and outputs as the MLP controllers described above, but differ in the size of its hidden layer, being 4, 8 and 16 units respectively. Two additional controller architectures were also based on recurrent networks but with impoverished inputs: the RMLP1WPOnly has only 6 inputs to its network, as angles and distances are given only to the current way point and the competitor's car, and the SimpleRMLP does not even input angle and distance to the competitor car to its network, which only has 4 inputs.

3) Modular controllers: The modular controllers represent an incorporation of domain knowledge into the controller architecture. The design is based on the observation that the most important task for a good controller, besides driving to a particular way point as quickly and reliably as possible, is to choose which way point to go for: the current or the next? Assuming that these two tasks are reasonably separable, the modular controllers are based on one MLP, that decides which way point to go for, and a SimpleRMLP controller that controls the car. The MLP receives three inputs: a bias, the distance to the current way point divided by the distance to the other car, and the speed of the car divided by the speed of the other car. If the only output of the MLP is above 0, the angle and distance to the current way point is fed to the SimpleRMLP, otherwise those of the next way point are fed.

Two versions of the modular architecture are tested: the ModularRMLP is initialised with all connection weights in both networks set to empty. The PrimedModular controllers, on the other hand, are initialised with a "blank" MLP but a SimpleRMLP that has already been solo-evolved to good fitness as an independent controller.

4) GP-based controllers: Finally, one controller architecture was based on genetic programming. Each controller consists of two function trees (evaluating to the two outputs, which are then interpreted as driving and steering) and three automatically defined functions (ADFs). The function trees are initialised randomly, and mutated with single-point macro mutation, where a randomly selected node in each tree is replaced with a randomly generated node. The trees are limited to a depth of 5 in order to make the computational expense of these controllers on par with the neural network-based controllers. When it comes to the node types, the set of terminals consists of sensor inputs (any of the eight inputs given to the MLP and recurrent controllers), constants (randomly initialised to values between 0 and 2) and ADF calls; the set of non-terminals consists of arithmetic functions (plus, minus, multiplication and protected division), trigonometric functions (sin, cos, tan and tanh) and an if-then-else function (if the first child evaluates to more than 0, return the value of the second child, otherwise the third child). To avoid loops, the ADF calls are restricted so that an ADF can only call an ADF with a higher id than itself.

C. Co-evolutionary algorithms

In this paper we compare two different co-evolutionary algorithms: generational and steady-state co-evolution.

1) Generational: We used a standard evolution strategy. For single population co-evolution, the steps are:

1. Evaluate each individual against another chosen at random from the best performing half of the population.
2. Pick the fittest half of the population to keep, and replace the other half with mutated versions of the fittest half.
3. Start another generation.

For multi-population co-evolution, a similar procedure is followed, only now each individual is evaluated against one of the fitter individuals from each population.

2) Steady-state: We use a modified version of the N-strikes-out algorithm, as detailed by Miconi and Channon. The one-population version consists of the following steps:

1. Pick two individuals A and B from the population at random.
2. Pit them against each other; determine the winner and the loser of the confrontation (if any).
3. If the loser has been defeated N times over its entire history, delete it and replace with a new individual.
4. Start another comparison.

In the two population case, Miconi and Channon found that a naive approach of comparing one random individual from each population caused disengagement, resulting in the weaker population losing any selection gradient.

They overcame this by comparing two individuals from population A against an individual from population B. The winner and loser were determined by which of the population A individuals scored best against the individual from population B. This process was then reversed, and two individuals from B were evaluated against an individual from A. The members of a population were thus only competing against each other, rather than other populations. We follow a similar approach in this paper, generalised to n populations.

One concern with the algorithm is that with a noisy fitness function, high fitness individuals can still be beaten occasionally by lower fitness individuals. Because individuals are deleted after a certain small number of defeats, this would mean losing desirable individuals. As suggested in [11], one way around this would be to introduce the concept of 'forgetting' old defeats.

The approach we took was to have a *defeat factor*, which we multiplied the number of losses by at each comparison. If less than 1, this should cause the number of losses to decay towards zero, being topped up by new losses. It means old defeats would be considered less important.

Another concern we had was that selection had no direct dependence on absolute fitness (an individual's score on the track). In theory, a controller could win many comparisons by blocking the other controller so it achieved a low score.

This would mean individuals that scored lower could still win lots of contests, and spread through the population.

This should be more of a concern with the single population N-strikes, but it could still be an issue in the multi-population case.

To combat this, we decided to make the defeat factor dependent on absolute fitness. We want individuals with a high apparent fitness to be given more evaluations before deletion, and low fitness individuals to quickly be replaced.

We therefore made the defeat factor $\mathcal{F}$ for an individual have the form:

$$\mathcal{F} = \frac{1}{e^{x_i - \bar{x}}} \tag{1}$$

Where x_i is the fitness of the individual, and $\bar{x}$ is the mean fitness of its population.

This means the defeat factor will be large in low-fitness individuals, and small for high-fitness individuals, implying more rapid forgetting of old defeats.

In order to adjust this parameter in a convenient manner, we introduced a defeat factor multiplier $\mathcal{D}$, as shown below:

$$\mathcal{F} = \frac{1}{e^{\mathcal{D}(x_i - \bar{x})}} \tag{2}$$

When $\mathcal{D}$ is 0, we get plain N-strikes behaviour. When $\mathcal{D}$ is greater than zero, we get an increasing contribution of absolute fitness.

One advantage of this form is that it retains the predictability of the number of deletions. In plain N-strikes, the number of deletions should be approximately the number of comparisons divided by N. We found that this behaviour was preserved by using this form of the defeat factor.

3) Parameter tuning: One of the features of the N-strikes out algorithm is its flexibility - there are many parameters that can be varied. In initial parameter tuning we tried varying the following parameters for the single-population case: population size P, number of strikes resulting in deletion N, the number of trials during an evaluation, and the value of the defeat factor multiplier $\mathcal{D}$.

Preliminary results suggested that a large population meant fitness initially rose slower but reached a higher value than in a small population. We also found small values of N performed better than larger values.

The only effect of changing the number of trials during an evaluation should be on the level of noise in the fitness function. We found that in high noise environments (such as using only one trial to compare two individuals), setting $\mathcal{D} = 0.5$ caused a quicker rise in fitness than the plain N-strikes, but both resulted in about the same fitness eventually.

For subsequent experiments, we chose the values of: $P = 30$, $N = 2$, at least 5 trials per comparison, and $\mathcal{D} = 0.5$.

The other decision to make was how to perform replacement. We used only mutation and not crossover in this paper, so the most straightforward choices were replacement by a mutated version of either the winner or loser of the comparison. We chose replacement by a mutation of the winner, as this should aid the rapid spread of high fitness through the population.

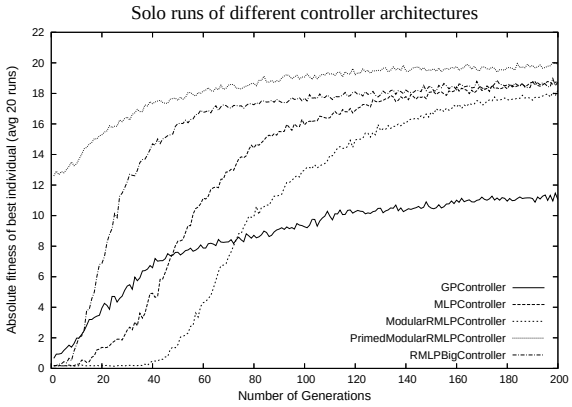

Fig. 1. Solo evolution of diverse architectures.

The generational algorithm had less parameters to adjust. We decided on a population size of 30, and at least 5 games per comparison, to allow direct comparison with the N-strikes-out algorithm.

III. Results

Our experiments proceeded as follows: first, to establish a baseline for the experiments with the many-population co-evolutionary algorithms, we tested each of the controller architectures independently. This was done using both solo evolution and single-population co-evolution. We then tried using both the generational and steady-state multi-population algorithms to compare controller architectures. A single controller architecture was then used to seed all populations of both multi-population evolutionary algorithms. The idea here was to investigate diversification between populations, and the extent to which multi-population competition helps evolve complex general behaviour.

A. Solo evolution of controller architectures

Our first set of experiments concerned the evolution of controllers for the single-car version of the task. All nine controller architectures described above were evolved in single-architecture populations using a standard $15 + 15$ evolution strategy for 200 populations. These experiments were all repeated for 20 runs. In figure 1 we have plotted the fitness growth of five different controller architectures. Several things can be learned from this figure: one is that the GP controllers consistently reach a much lower final fitness than the other architectures, even though they learn quite fast in the first few generations. The PrimedModular controllers perform slightly better than the others in the end, but this is probably because they have effectively evolved for longer, the lower layer being pre-evolved. Other than that, RMLPBig (large recurrent network) learns fastest of the controllers.

In figure 2 we compare the five different controller architectures based on monolithic (non-modular) RMLPs. What is remarkable here is how similar their ultimate performance is. The only difference of any note is in their learning speed, and here we see a clear relation to the size of the network: the larger the network is, the faster it learns.

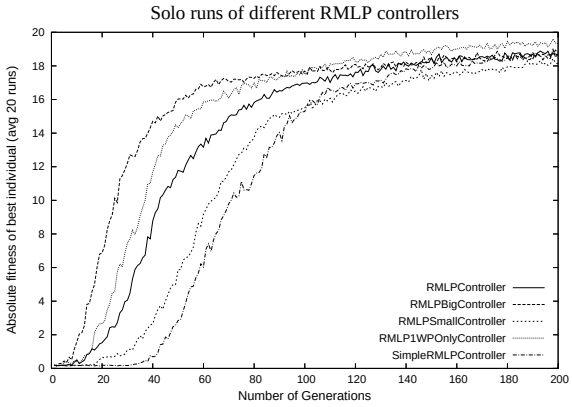

Fig. 2. Solo evolution of different rmlp-based architectures.

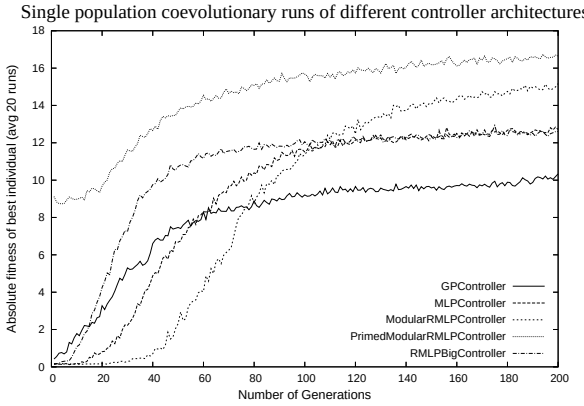

Fig. 3. Single-population generational co-evolution of diverse architectures.

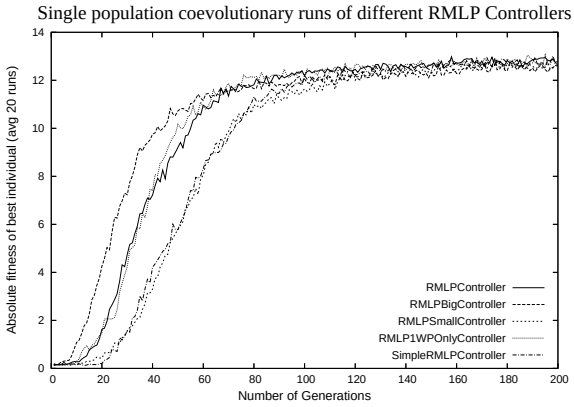

Fig. 4. Single-population generational co-evolution of different rmlp-based architectures.

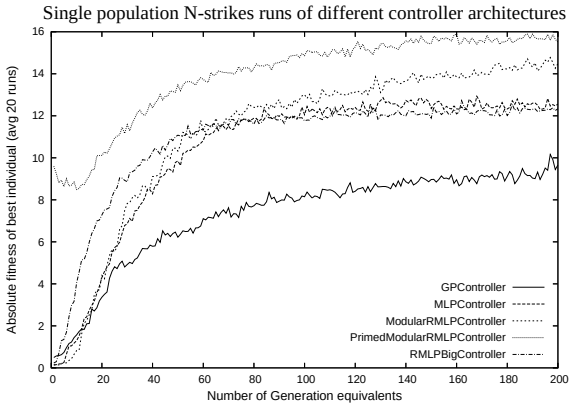

Fig. 5. Single-population steady-state co-evolution of diverse architectures.

B. Single-population co-evolution

Next, we ran a number of experiments where we co-evolved controllers for the two-car version of the task, using single-population co-evolution with populations of 30 individuals containing only one controller type each. So in these runs, the controllers only ever compete against other controllers of the same architecture. We did 20 runs for each architecture, using both the generational and the steady-state co-evolutionary algorithms.

1) Generational: In figure 3 we plot the fitness of the best controller of each generation for the same five controller architectures as in figure 1. As in the solo evolution, the GP controllers start out as fast learners but are by far the worst of the lot at the end of 200 generations. Unlike in the solo runs, we see a very clear superiority of the modular controllers over the other architectures. At the end of the runs, the primed modular controllers do better than the non-primed, but the fitness for the non-primed ones is still increasing.

Looking at the RMLP-based controllers (figure 4) we see no difference in ultimate fitness and little difference in learning speed. The larger networks learn somewhat faster.

2) Steady-state: The results of the steady-state runs were very similar, as can be seen from figure 5 (we have omitted the graph for rmlp-only comparisons out of space considerations). The primed modular controller still performs almost twice as well as the GP controller. One difference is that the non-primed modular controller learns much faster with steady-state than with generational co-evolution.

C. Multi-architecture multi-population co-evolution

And so, at last, we come to the multi-population co-evolution. We used nine populations for these experiments, one population for each controller architecture. In these runs, which lasted for 500 generations (or steady-state equivalents) each individual of each population is tested against individuals of all other populations, thus controllers of all other architectures. At least 20 runs were made for both the generational and steady-state algorithms.

1) Generational: Figure 6 plots the fitness growth of the populations populated by the main different controller architectures (remember that all nine populations were part of the runs, though only five are plotted). Some of what we see here could be expected, given our single-population results. The modular controllers still win (literally) over the other controllers. But what is unexpected is that the GP-based controllers no longer do worst, indeed they perform almost as well as the MLP-based controllers at the very end of the 500 generations. Instead, the RMLPBigControllers perform worst by a significant margin.

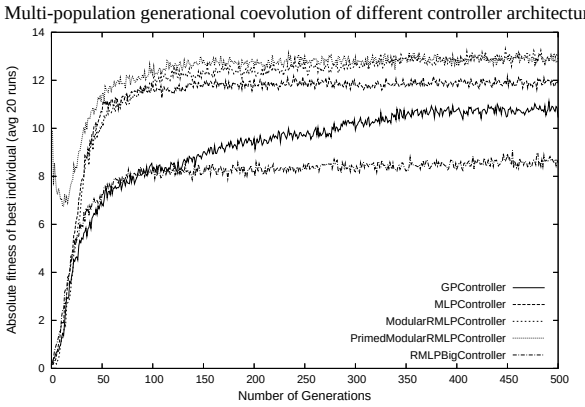

Fig. 6. Nine-population generational co-evolution of diverse architectures.

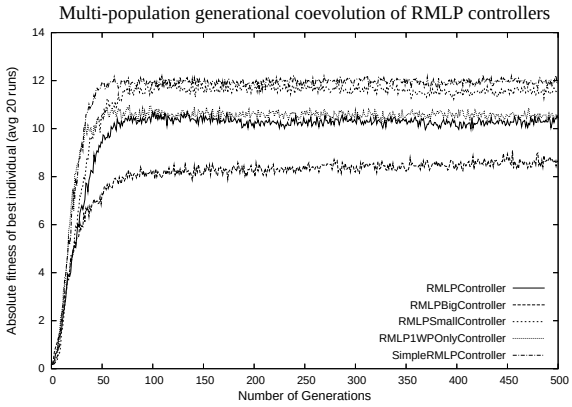

Fig. 7. Nine-population generational co-evolution of different rmlp-based architectures.

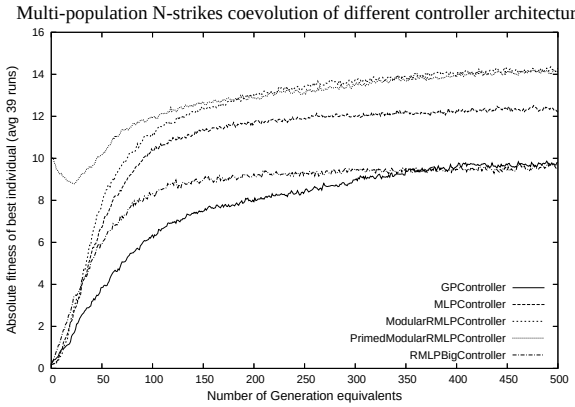

Fig. 8. Nine-population steady-state co-evolution of diverse architectures.

Controller	solo	heuristic	fixed	competition score
n-pop gen MLP	16.86	12.01	12.27	13.71
n-pop gen Modular	16.49	9.52	**13.37**	13.13
n-pop nstr MLP	17.40	**11.85**	12.73	**13.99**
n-pop nstr Modular	16.59	10.54	12.96	13.36
1-pop gen Modular	16.38	6.96	13.20	12.18
1-pop gen RMLPBig	14.90	11.30	11.89	12.69
1-pop nstr Modular	15.19	10.06	10.16	11.80
1-pop nstr RMLPBig	16.64	11.08	11.68	13.13
Solo Modular	**18.50**	2.80	3.20	8.17
Solo RMLPBig	17.35	4.91	6.54	9.60

TABLE I
COMPETITION SCORE OF A VARIETY OF HIGH-PERFORMING CONTROLLERS

This rather surprising result concerning the RMLPBig controller architecture is put into context when looking at figure 7, which compares only the RMLP-based controllers. Here we see that the smaller the RMLP-based controller is, the better ultimate fitness it reaches, with the minimalist SimpleRMLPControllers coming out on top.

2) Steady-state: The steady-state runs paint a similar picture. The main difference between figure 8, which shows the fitness growth of the main controller architectures, and figure 6 is a slower overall fitness growth, and that the GP controllers as a result don't do any better than the RMLPBig controllers, i.e. not very good at all.

The graph for the RMLP-based controllers has been omitted in order to conserve space, but looks essentially like figure 7 but with slower fitness growth.

D. Single-architecture multi-population co-evolution

In the final set of experiments, we filled all nine populations with RMLPBig controllers, as these are in theory capable of expressing the most complex strategies. We then evolved them for 500 generations, 10 runs each for the generational and steady-state algorithms. We are not showing any graphs of the same type as for the other experiments, as these would be quite uninteresting: all the populations reach the same fitness (of their best individuals) on average, with little differences between the populations in an individual run. This fitness is around 8, virtually the same as the fitness of the RMLPBig controllers in the multi-architecture runs.

E. Comparison of best controllers

At this point, the reader will probably wonder which of the experiments above actually produced the best controllers. As always with co-evolution, this question is not straightforward to answer. But we've tried to answer this in two ways: by measuring the competition score of representatively high-performing controllers (the best we could find from a limited probe) from each experiment, and by testing the same controllers against each other in two-car racing.

Competition score is the score used to rank submitted controllers in the league table of the CIG Car Racing Competition. It is calculated by letting the controller race 500 trials on its own, and 500 trials each against a rather low-performing hard-coded heuristic controller and a medium-performing MLP-based controller. Table I shows the competition score breakdown for the selected high-performing controllers. Judging from these scores, the multi-population runs yielded the best overall controllers, the single-population runs slightly less good and the solo evolutionary runs really rather bad controllers. The best controller found seem to be based on a MLP, but the differences are not great between the architectures.

	n-pop gen MLP	n-pop gen Modular	n-pop nstr MLP	n-pop nstr Modular	1-pop gen Modular	1-pop gen RmlpBig	1-pop nstr Modular	1-pop nstr RmlpBig	Solo RmlpBig
n-pop gen MLP	0.00	-0.75	0.09	-0.42	-0.81	1.20	2.37	1.17	4.58
n-pop gen Modular	0.75	0.00	0.71	0.35	-0.09	1.91	2.65	1.90	5.48
n-pop nstr MLP	-0.09	-0.71	0.00	-0.53	-0.31	1.08	2.81	1.50	4.64
n-pop nstr Modular	0.42	-0.35	0.53	0.00	0.17	1.06	2.82	1.52	4.31
1-pop gen Modular	0.81	0.09	0.31	-0.17	0.00	2.10	0.68	1.77	3.57
1-pop gen RmlpBig	-1.20	-1.91	-1.08	-1.06	-2.10	0.00	1.56	0.36	4.15
1-pop nstr Modular	-2.37	-2.65	-2.81	-2.82	-0.68	-1.56	0.00	-1.51	8.02
1-pop nstr RmlpBig	-1.17	-1.90	-1.50	-1.52	-1.77	-0.36	1.51	0.00	3.88
Solo RmlpBig	-4.58	-5.48	-4.64	-4.31	-3.57	-4.15	-8.02	-3.88	0.00

TABLE II

SCORE DIFFERENCES IN COMPETITIONS BETWEEN CONTROLLERS (AVERAGE OF 500 GAMES). A POSITIVE VALUE MEANS THAT THE CONTROLLER OF THE CURRENT ROW BEATS THE CONTROLLER OF THE CURRENT COLUMN.

Table II shows the results of direct competition between these controllers. The differences are sometimes quite dramatic, as when the solo-evolved RMLPBig gets 8 points lower fitness than the single-population-evolved modular controller. If a clear winner has to be picked, it is the modular controller evolved with generational multi-population co-evolution, which does not lose significantly to any other controller. Generally, the dominance pattern of multi-population over single-population over solo evolution persists.

IV. CONCLUSIONS

We set ourselves a handful of questions to solve at the beginning of the paper, and even though we can draw quite a few conclusions from our experiments, all the questions have not been answered. We have seen that when two cars were involved, the modular controllers outperform all the other controller architectures. For solo- and single-population co-evolution, larger controllers learn faster, whereas this is not always the case for multi-population co-evolution. It thus seems that to the extent that multi-population co-evolution is useful for comparing the performance of different controller architectures, it is so in a rather different way than single-population co-evolution. However, multi-population co-evolution produces generally better controllers than either solo evolution or single-population co-evolution, corroborating one of our main hypotheses.

What we have not had time (and space) to investigate here is the behavioural diversity between the evolved controllers from the different experiments. Thus, we don't know whether the superior performance of the multi-generation co-evolution is because of increased diversity between populations, but we still think this is the case.

A major unsolved puzzle is why the RMLP-based controllers did better the simpler they were, in the multi-population experiments. Our main hypothesis here is that because the more complex controllers learn faster, they settle for a particular strategy sooner than the others, and that this particular strategy is not a very good one because the other controllers have not yet developed any useful strategies. These strategic niches function as local optima, as learning a generally better strategy would require first unlearning the current mediocre strategy. The slow-learning simple controllers are instead forced to learn more general strategies. If this is true, and the effect generalises to other problems, this phenomenon could enhance our understanding of co-evolutionary dynamics.

REFERENCES

[1] D. B. Fogel, *Blondie24: playing at the edge of AI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[2] M. Parker and G. B. Parker, "The evolution of multi-layer neural networks for the control of xpilot agents," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[3] J. Togelius and S. M. Lucas, "Evolving controllers for simulated car racing," in *Proceedings of the Congress on Evolutionary Computation*, 2005.

[4] ——, "Evolving robust and specialized car racing skills," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[5] ——, "Arms races and car races," in *Proceedings of Parallel Problem Solving from Nature*. Springer, 2006.

[6] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation in racing games," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[7] J. Togelius, R. De Nardi, H. Marques, R. Newcombe, S. M. Lucas, and O. Holland, "Nonlinear dynamics modelling for controller evolution," in *Proceedings of GECCO*, 2007.

[8] R. Calabretta, A. Di Fernando, G. P. Wagner, and D. Parisi, "What does it take to evolve behaviorally complex organisms?" *BioSystems*, 2002.

[9] J. Bongard and C. Paul, "Making evolution an offer it can't refuse: Morphology and the extradimensional bypass," in *Proceedings of the European Conference on Articial Life*, 2001.

[10] C. Rosin and R. Belew, "New methods for competitive coevolution," *Evolutionary Computation*, vol. 5, no. 1, 1996.

[11] T. Miconi and A. Channon, "The n-strikes-out algorithm: A steady-state algorithm for coevolution," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006, pp. 1639–1646.

[12] D. H. Janzen, "When is it co-evolution?" *Evolution*, vol. 34, no. 3, pp. 611–612, 1980.

[13] S. Bullock, "Co-evolutionary design: Implications for evolutionary robotics, Tech. Rep. CSRP384, 1995.

[14] G. S. Hornby and B. Mirtich, "Comparing diffuse and true coevolution in a physics-based world, Tech. Rep. TR-98-11, 1999.

[15] J. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.