Evolving Controllers for Simulated Car Racing using Object Oriented Genetic Programming

Alexandros Agapitos
University of Essex
Dept. of Computer Science
Colchester, CO4 3SQ
United Kingdom
aagapi@essex.ac.uk

Julian Togelius
University of Essex
Dept. of Computer Science
Colchester, CO4 3SQ
United Kingdom
jtogel@essex.ac.uk

Simon M. Lucas
University of Essex
Dept. of Computer Science
Colchester, CO4 3SQ
United Kingdom
sml@essex.ac.uk

ABSTRACT

Several different controller representations are compared on a non-trivial problem in simulated car racing, with respect to learning speed and final fitness. The controller representations are based either on neural networks or genetic programming, and also differ in regards to whether they allow for stateful controllers or just reactive ones. Evolved GP trees are analysed, and attempts are made at explaining the performance differences observed.

TRACK: GENETIC PROGRAMMING

Categories and Subject Descriptors

I.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming

Keywords

Evolutionary robotics, neural networks, Object-Oriented genetic programming, Subtree macro-mutation, Homologous uniform crossover

1. INTRODUCTION

An important open question for evolutionary robotics (ER), and evolutionary game AI, is how best to represent the controller. Currently, most controllers in evolutionary robotics are represented as neural networks of some sort, either feed-forward multilayer perceptrons (MLPs) or some sort of recurrent network, such as Elman-style networks or CTRNNs. The former have the advantage of simplicity and well-understood theoretical properties, and the latter have the advantage of (potential for) integration of information over time, thus allowing for deliberative rather than just reactive controllers.

Puzzlingly, very little work in evolutionary robotics seem to make use of the vast knowledge accumulated in the sister field of genetic programming (GP), which deals with the evolution of computer programs represented in some symbolic form. It is not at all clear why this is so. For one thing, the investigation of stateful versus stateless controllers in ER is closely mirrored by the ongoing investigation of how to best evolve programs with state in GP. We believe there could be much fruitful interplay between these fields.

In this paper, we compare neuroevolution and GP, both with and without state, on some variations of a simulated car racing problem. We hope to be able to understand some of the similarities and differences between those two families of controller representations on problems of agent control in robotics and computer games.

1.1 Evolutionary car racing

The problem of racing a simulated car around a track is interesting from several perspectives. From an applications perspective, car racing in its myriad of both simulated and real forms is an ever-popular form of entertainment, and the problem of getting a vehicle from point A to point B as fast as possible can hardly be said to be without practical relevance even outside of entertainment. There thus exists ample application potential for methods for optimising various aspects of this approach.

From the perspectives of machine learning, and of evolutionary robotics, the problem of how to win a car race is far from a solved one and thus the problem of learning how to win a car race is even further from being solved. The task has a certain appeal to the evolutionary roboticist, in that while it is fairly easy to learn to navigate a simple track by driving slowly and keeping your distance to the walls, beating good competitors in a multi-car races on a varied selection of challenging tracks requires considerable training and a diverse skillset. These skills would have to include modelling the dynamics of the driver's and opponents' cars in various situations, modelling the competitors' driving style and epistemic state, navigating complex environments, planning (e.g. when to overtake and go for the pit stop) and other high-level cognitive skills as well as just fast and accurate reactions.

It has been previously investigated how to best evolve controllers for single-car, single-track racing [8], how to generalise controllers to reliable drive on several dissimilar tracks and specialise them for particular tracks [10] and the impact of fitness functions on competitive co-evolution of two cars on the same track [9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

The main goal of the evolutionary car racing project, however, is still to find methods for evolving controllers with good, general driving skills. One aspect of this project which has so far received inappropriately little attention is the controller representation. Almost all of our controllers have been represented as small, feed-forward multilayer perceptrons. In this paper we are therefore comparing this representation with larger feed-forward neural networks, recurrent networks, and several types of genetic programming.

1.2 Motivation for applying Genetic Programming

The vast majority of evolved programs use a functional tree representation and while GP has produced some impressive results it has significant problems with scalability. Most GP evolved programs are simple expression trees that perform simple mappings from inputs to desired outputs. Even since the addition of effective storage and retrieval of arbitrarily complicated state information to GP, limited research has been devoted to the evolution of programs that utilize state variables. Many interesting problems require a program to preserve some sort of state in-between its computations. The notion of state is a very important concept used by human programmers as means of naming semantically important features that can be used multiple times or that describe a self-contained entity. The use of state can come in many different incarnations - be it a single local/global variable, an arbitrary data structure, up to a point of an encapsulated collection of data that is being exclusively operated upon by a set of methods, which naturally leads to data abstraction. Nevertheless, it seems unlikely that a general-purpose memory manipulation capability will prove to be advantageous to the evolved solution of an arbitrary problem. Stated differently, not all programs that use memory will have a significant advantage over programs that ignore their memory or that do not have memory, for an arbitrary problem. Heading towards the direction of evolving complete classes and co-operating sets of classes we need to study (a) the nature of problems and the circumstances under which the evolved solutions can enjoy an evolutionary advantage from using state variables; (b) the way in which the evolutionary process decides which features to keep track of and how the storage/retrieval of intermediate state can facilitate the evolvable computation; (c) the impact that the use of state-manipulation primitives has on the search operators for exploring the space of constructible programs; (d) the effect of state-manipulation primitives to program representation and interpretation.

From an implementation point of view, hand-coding a controller that will proficiently race a car around a repertoire of tracks is a challenging, open-ended problem. Simply stated, a perfect solution is often theoretically impossible and the nature of the problem renders the definition of analytical algorithms problematic. As an example, imagine the set of conditions in which a car should or should not steer when it is moving based on sensory input. It was felt that many lessons could be learned by allowing the solution to be provided by an automated program induction technique, such as GP. The intension was to evolve a program for a problem whose solution is not precisely known in advance.

However, there are many possible implementations and these are governed by the programming space that a human programmer or a program-generation engine are allowed to operate on. This space is concerned with the program representation given a specific primitive language. The use of state-manipulation constructs defines a different programming space, than the one where no state information is exploited, and allows programs to store and retrieve information on various features that they choose to be useful during their computations. We would like to see whether the use of state adds any significant advantage to the evolution of a proficient controller. If the evolutionary process finds some workable representation of the useful environmental features, it may be the case that these can be maintained and manipulated as state information in order to drive more efficiently. So at any given time a state-aware controller can use the information provided by the sensory input and that which builds and maintains on its own. In addition, it would be interesting to see which features are chosen to represent the object state space. Our discussion will be laid in an OO context so the terms memory and object state space will be used interchangeably.

```
public interface CarController{
    public double[] drive(SensorModel sm);
}
```

Figure 1: The interface specifying the signature of the driving method

1.3 Object-Oriented Genetic Programming

1.3.1 OO vs. Functional Programming Spaces

The output of the car controlling program is an array of two real values, the first being interpreted as the driving command whereas the second as the steering command. Figure 1 presents the signature (return type and parameter types) of the interface method `drive()` that is used as a contract between the evolved program and its clients. The parameter of type `SensorModel` provides the environmental input which is discussed in a later section. As with most programming problems, there are many possible implementation routes and we can encourage the EA to induce a specific implementation by allowing it to work on a particular programming space. When the program that implements the `CarController` interface has been constructed in an OO programming space, it is allowed state variables along with methods that inspect and modify this internal state. A crucial aspect of state-aware programs is that they often exhibit time-dependent behavior. That is, the order in which the program stores and retrieves state information can have a concomitant impact on its output. This particularity can introduce many dependencies upon various program parts (i.e among different functions) that manipulate state information in an explicit order, and in our opinion is one of the main reasons that hinders the evolution of programs that use memory. Very importantly, the program must evolve to ensure that the storage is written to before it is read and that the manipulation of state information by various program parts is performed in a compatible way. On the other hand a stimulus-response program that sits in a functional programming space will not enjoy the use of state information.

1.3.2 Evolvable Class Representation

Following previous work on the evolution of complete classes [1] we decided to represent an evolvable individual using a syntactic structure that couples a linear repository of *class* and *instance variables* representing the object state along with a set of *evolvable methods* (using an expression tree representation) that are responsible for the way an object acts and reacts, in terms of state changes and message passing.

1.3.3 State Representation

The use of memory in GP dates back to the work of Koza (1992), who used global registers that could be manipulated with specially built storage operators. Teller (1994) introduced *Indexed Memory* to allow a selection from an arbitrary set of memory cells [7]. Additional `read` and `write` operations are made available in the language to allow this memory to be accessed and manipulated by various program parts. Koza (1999) went on and generalized both the above to the notion of an *Automatically Defined Store* [4]. Finally, Kirshenbaum (2000) presented work on the evolution of programs that use statically scoped local variables [2]. That is, variables whose visibility is bounded to a given scope - a subtree rooted on a `Let` construct.

We employed a simple memory addressing scheme by combining *type* information along with the mechanism of *pass by reference* [5] in order to operate on the object state space. The object memory is represented as a linked list of objects of interface type `Settable`, reminiscent of Teller's indexed memory but uses a different way to store and read values. References to these `Settable` objects are added to the language used to construct programs, and these represent the available instance variables (object state space). Within a program structure these language elements are being passed by reference to specially built primitives that explicitly set the value of their argument. Once the method returns, the value of its argument will have been updated. For our purposes a `setValue(Settable s, double value)` primitive method has been defined. Notice that the use of *strong typing* for drawing a distinction between settable variables and their underlying values allows for the emergence of various sorts of assignment schemes and obviates the need of devising a strategy to deal with illegal range of index values, a substantial issue when working with traditional indexed memory. We follow Teller's example and `setValue` is defined to return the original value held in the `Settable` object it has just overwritten.

1.3.4 Variation Operators

Our search employs two different variation schemes. These are *Subtree macromutation* (MM – substituting a node in the tree with an entirely randomly generated subtree of the same return type, under depth or size contraints) and homologous *Uniform Crossover* (UXO) along with *Point Mutation* (PM) as defined by Poli and Langdon [6]. In the case of multi-tree programs the evolutionary algorithm must come to a decision as to which tree the variation operator will be applied. Each time the variation operator will be applied to the expression tree implementing the interface method `drive` with a probability of 1.0 and to each supplementary expression tree with a probability of 0.5. Additionally, for MM, other than choosing the tree node to be replaced at random we devised an additional simple node selection scheme that allows us to select nodes at different depth levels using a uniform probability distribution, with the expectation to render bigger changes more likely.

1.3.5 Program Representation Language

We defined a diverse set of language elements to form a general programming space for the evolutionary algorithm to work on. This is presented in table 1. Standard arithmetic operators have been provided (`add`, `sub`, `mul`, `div`) along with state-manipulation operators (`setValue`), predicates (`>`,`>=`,`=`,`<`,`<=`) and an `IF-Then-Else` construct that allows to control the flow of execution within the program such that every expression tree rooted at that node will be interpreted using *lazy evaluation*. The program is required to return an array of two `double` values so `rootGlue` has been defined as a wrapper that accepts two `double`s and returns a `double` array populated with these argument values. The car controller receives environmental input using four different sensors. These are modeled as method invocations on a `SensorModel` object (see figure 1). The `wallSensorReading` method requires two parameters of type `double` that specify the *angle* and the *range* of the sensor. The range of the sensor is equal to the range parameter multiplied by 200 pixels; this parameter is constrained to be within the [0, 1] interval. For example, `wallSensorReading(`π`/2, 0.75)` returns an estimation of the distance to the wall along a line protruding straight to the left of the car, as a proportion of 150 pixels. If the first wall to the left of the car is 100 pixels away, the `wallSensorReading` method will return around 0.66 in the example given. Method `speed` returns the driving behavior and it's 0.3 for driving forward, 0.0 for neutral and -0.3 for backwards. The `angleToNextWaypoint` method returns the difference between the current orientation of the car and the angle between the center of the car and the next waypoint. Similarly, `distanceToNextWaypoint` returns the distance between the center of the car and the next waypoint. All angles are unwrapped and a small amount of gaussian noise is added to all readings.

2. METHODS

2.1 Car simulation and fitness measure

This simulation, which is intended to qualitatively model driving a radio-controlled toy car on a tabletop racing track, has a car with dimension 20*10 pixels driving on a 400*300 pixels racing track. While the simulation is based on a reasonably realistic physics model, allowing for momentum, collisions, and skidding, it does not behave identically to any particular physical system. At each time step (the simulation is updated at 20hz in simulated time) a command is sent from the controller to the simulation, which executes the command and returns the new state of the car.

Selected elements of the state are available to the controller via an interface. The available information is all such that it could in principle have been gathered by sensors places on the car, and is as such "first person": speed of the car, angle and distance to the next way point and distance to the wall in a given direction relative to the heading of the car. A small amount of noise is added to all sensor readings.

As for the outputs of the controller, these are two real numbers which are interpreted by the simulation as any of nine possible commands - typical toy cars of the sort that inspires our simulation have bang-bang control, hence the

Table 1: Primitive elements for evolving car controlling programs

Method set		
Method	**Argument(s) type**	**Return type**
rootGlue	double, double	double[ ]
State Manipulation		
setValue	Settable, double	double
Sensory input		
wallSensorReading	double, double	double
speed	-	double
angleToNextWayPoint	-	double
distanceToNextWayPoint	-	double
Arithmetic		
add	double, double	double
sub	double, double	double
mul	double, double	double
div (protected)	double, double	double
Predicates		
$>$, $>=$, $=$, $<$, $<=$	double, double	boolean
Conditional		
IF-Then-Else	boolean, double, double	double
Terminal set		
Terminal	**Value**	**Type**
Constants	π, $-\pi$, $\pi/6$, $-\pi/6$, $\pi/12$, $-\pi/12$ -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0	double
Perameters	subject to environment model of evaluation	double

discretisation. The first controller output is interpreted as the command for driving forward if its value is above 0.3, backward if below -0.3 and neutral otherwise. The second output is interpreted as steering left, right or centre in the same manner.

The fitness of a certain controller is given by how far around one or several tracks it manages to travel in 700 time steps. This is measured by how many way points it manages to pass within that time. Each racing track is defined by a starting position, a set of walls and a chain of way points. The eight tracks used for the experiments in this paper are depicted in figure 2. A way point is considered to be passed if the centre of the car is within 30 pixels of the centre of the way point, and that way point is the next one in the chain (way points must be passed in order). Passing all way points within 700 time steps yields fitness 1, passing fewer way points leads to lower fitness and completing several laps of the track within the allotted time gives higher values. The maximum possible fitness for most tracks seem to be between 2 and 4.

2.2 Evolutionary algorithm

For ease of comparison, the same evolutionary algorithm is used for all presentations. This algorithm is a 50+50 Evolution Strategy, meaning that each generation the 50 best controllers are retained, while the 50 worst are deleted and replaced with mutated copies of the 50 best. (The exact mutation mechanism is detailed in the sections on the respective controller representations below.) To alleviate the considerable noisiness of the fitness evaluations, the fitness of each controller is calculated as the average of three independent trials.

2.3 Evolving functional controllers

For the functional representation we used both standard GP and GP with *ADFs* using the *Evolutionary Selection of Program Architecture* method as it is described in [3]. Each functional program is allowed a maximum of 5 ADFs and each ADF is allowed a maximum of 3 arguments. Parameter and return types are set to `double` to be compatible with the numeric primitives present in the programming space. Similarly to [3], a simple ADF naming scheme is employed is order to prevent the emergence of circular calling dependencies among the functions which can result in non-halting programs. This would require the experimenter to handcraft a way of stopping their evaluation and assign them fitness - an issue clearly out of the scope of this paper. This was done to place a reasonable limit on the duration of the evolutionary run. When limited in this way each individual run still took approximately 7 hours on a 3.4GHz P4 machine.

Under this representation the program returns its output via the expression tree that is explicitly dedicated to the interface method `drive` in figure 1. In the presence of ADFs the representation system is enhanced and GP is allowed to co-evolve additional modules advantageous to the composition of the final solution. No restrictions were placed upon which primitives can be used by which function-defining expression tree of a multi-tree program, thus the function and terminal sets were identical both for method `drive` and any supplementary ADFs. State is not allowed, so the language used for the functional representation contained all elements of table 1 but `setValue`. Respectively, no `Settable` objects are added to the terminal set.

2.4 Evolving object-oriented controllers

The genome representation in the case where we evolve a complete class has been outlined in section 1.3.2. During the generation of the initial population the EA performs a random sampling of *Evolvable Class* structures. The generation of such an individual is analogous to the process of initialising an individual using the *Evolutionary Selection of Program Architecture* method with the addition of randomly selecting the number and type of instance variables. Analogously to section 2.3 the number of additional instance

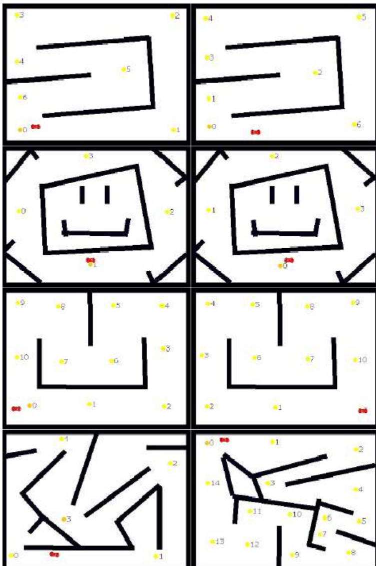

Figure 2: The eight tracks. 5 and 6 differ in the clockwise/unti-clockwise layout of waypoints

methods is set to the random interval of [1, 5] (using a uniform probability distribution) and the number of the argument that each one may possess is set to the random interval of [1, 3]. The range of potentially useful numbers and types of instance variables cannot be predicted with certainty for an arbitrary problem. Here we require the number of instance variables to be uniformly drawn from within the [5, 15] interval. In addition, the type of information that can be stored in the form of object state is set to be of type `double` in order to be compatible with the numeric input and output types of the primitive language elements. Notice that once the number and type of instance variables are specified, they cannot be altered by the application of variation operators.

Similarly to section 2.3, no circular calling hierarchies are allowed, and no restrictions are being placed as to which primitives can be used by which expression tree. State manipulation is allowed so `setValue` and a number of `Settable` objects are made available in the primitive language.

The fitness evaluation of state-aware programs begins with the initialization of object state variables at time step t_0. We do not allow the evolution of explicit constructor methods. Alternatively, we require that all numeric instance variables be set to zero. This is in-line with most modern OO programming languages, - Java for example will implicitly set the values of numeric instance variables to zero in the absence of a constructor method. Hence, at each timestep t_n the object is being operated upon by invoking the interface method `drive` with the state variables being set to the value that was stored at time-step t_{n-1}. It is noteworthy that the fitness evaluation simulates the life-cycle of a *passive* object that is being born when its state variables are initialised to zero, it is being acted upon by invocation to the interface method `drive` for 700 time-steps, and finally dies along with the cease of fitness evaluation procedure.

2.5 GP Run Parameters

We defined a diverse set of evolutionary run parameters depending on the representation employed and the variation operator used. All initial generations are being populated using the *Ramped-Half-and-Half* algorithm with a maximum depth constrained by a `maxInitialDepth` parameter. For standard GP (program with a single result producing branch) the maximum initial depth when using MM was set to 5 and expression trees were allowed to grow up to depth of 10. In the case of UXO [1] both the initial and maximum depths are set to 10. For GP with ADfs and OOGP the maximum initial depth for the case of MM was set to 5 whereas the maximum tree depth was set to 8. Analogously, for UXO both initial and maximum depths are set to 8.

In the case of macro-mutation search regime, MM is the sole operator and is applied with 100% probability. For the recombination search regime, HXO is applied with 100% probability and subsequently each offspring is subjected to PM with a variable mutation probability p_m. This is induced by dividing 2 by the size of the tree and refers to the probability that a single node will be mutated.

2.6 Evolving neural network controllers

In the neural network controller representation, each controller is based on either a multi-layer perceptron or on a Elman-style recurrent neural network (essentially an MLP with an added layer of connections to the hidden neural layer from the hidden layer of the last update of the network). All the MLPs take the speed and the angle to the next waypoint as inputs, and in addition to this they take a number of wall sensor readings as inputs as determined at the start of each experiment. These wall sensors are constant throughout the evolutionary run, and are distributed at roughly even angular intervals around the car (accessed through calls to getWallSensorReading with a constant angle and a range of 1).

We evolve three different neural network configurations: MLPs with 12 hidden neurons and 12 wall sensors, Recurrent neural networks with 12 hidden neurons and 12 wall sensors, and MLPs with 12 hidden neurons but only 4 wall sensors.

3. RESULTS

3.1 Single-track controllers

The first set of experiments concern the evolution of driving skills on a single track. The track in question is track 5 on figure 2, which is one of the easiest of the eight tracks. In this set of experiments, each evolutionary run starts with freshly created controllers: neural networks with all connection weights set to 0, or GP controllers with small randomly generated trees. For each controller configuration, we ran 10 independent run of 200 generations with population 100. The results of this can be seen in table 2.

The first observation on the results is that all configurations of both NN and GP controller representations managed to evolve high-performing car drivers. But the performance of GP and NN controllers are not identical. Generally, it can be seen that the GP controllers evolve much faster than the NN controllers, but that the NN controllers ultimately

[1] A very interesting property of UXO is that the offspring will not grow past the depth of its deeper parent and thus past the depth of the deepest program populating the initial generation.

Table 2: Average fitness of best controller for generations 10, 50, 100, 200 (averaged over 10 independent evolutionary runs - std. deviation in parentheses)

Method	10	50	100	200
Functional/no ADFs/Macromutation	1.26 (0.65)	2.33 (0.4)	2.47 (0.4)	2.51 (0.15)
Functional/ADFs/Macromutation	1.54 (0.45)	2.54 (0.17)	2.62 (0.15)	**2.67** (0.1)
Functional/no ADFs/Recombination	1.87 (0.52)	2.38 (0.16)	2.45 (0.17)	2.46 (0.17)
Functional/ADFs/Recombination	1.62 (0.74)	2.23 (0.63)	2.39 (0.47)	2.53 (0.17)
OO/Macromutation	1.55 (0.61)	2.47 (0.7)	2.54 (0.18)	2.59 (0.18)
OO/Recombination	1.10 (0.75)	2.39 (0.3)	2.47 (0.07)	2.55 (0.07)
MLP	0.13 (0.17)	2.48 (0.67)	2.92 (0.09)	**3.08** (0.07)
Recurrent	0.19 (0.16)	1.06 (0.45)	2.43 (0.46)	2.92 (0.16)
MLP with less sensors	0.19 (0.22)	2.65 (0.08)	2.94 (0.08)	3.07 (0.02)

reach higher fitnesses. No significant difference can be seen between functional and object-oriented GP, and between recurrent and feedforward neural nets.

3.2 Performance on several tracks

Next, we looked at the generalisation capability of the controllers evolved in the preceding section. This was done by trying each of these controllers on each of the eight tracks, not only the track for which they had all been evolved.

Out of space considerations, we have omitted the table detailing the results, but it can safely be said that in general, the evolved neural networks perform significantly better than the GP controllers even for tracks for which they have not been evolved. Like in the previous section, not much of a difference was found between stateful and stateless controllers. And like in our previous paper [10], all controllers scored lower on other tracks than on track 5, for which they had been evolved, but scored slightly better on tracks which like track 5 run counter-clockwise than on those that run clockwise.

3.3 Evolving generalisation

The final set of experiments concern the incremental evolution of general controllers. In these experiments, each evolutionary run was seeded with the results of the 200 generations of evolution on a single track described above. Then evolution proceeded for 50 generations, with the crucial difference that the fitness function was made incremental; each controller evaluation was done as the average of the progress that controller displayed on a set of tracks. At the first generation of each evolutionary run, only track 5 was used for fitness evaluations, but every time a controller reached fitness 1.5 a track was added to the evaluation set (and kept for the duration of the evolutionary run). The sequence in which tracks were added was 5, 6, 3, 4, 1, 2, 7, 8. (Note that every second track added runs clockwise instead counterclockwise, to increase the diversity between tracks added in sequence and thus avoid overfitting to a particular driving direction.) A controller that was able to generalise completely and drive proficiently on all tracks would at the end of these 50 extra generations have all 8 tracks in its evaluation set, whereas a very poor generaliser would be stuck with only track 5 in the set.

As a consequence, the graphs for these evolutionary runs include not only the fitness of the best controller in the population but also the incrementation level (number of tracks in the evaluation set) of the population.

As we can see from table 3, both neural network and genetic programming are able to incrementally generalise previously evolved controllers and achieve proficiency on a majority of the eight tracks. However, there is a marked difference, in that the neural network-based controllers on average generalised significantly better, and end up being able to drive proficiently on seven of the eight tracks.

Table 3: Average number of tracks, proficiently driven (averaged over 10 independent evolutionary runs - std. deviation in parentheses)

Method	Avg. no of tracks
Functional/ADFs/Macromutation	6.0 (0.8)
OO/Macromutation	5.8 (0.91)
OO/Recombination	4.6 (1.07)
MLP	8.0 (0.0)
Recurrent	8.0 (0.0)
MLP with less sensors	8.0 (0.0)

On the other hand, there seems to be no significant performance difference between stateful and stateless controllers, i.e. between mlps and functional gp on the one hand and recurrent networks and oogp on the other hand. Just like in the results above.

3.4 A Gallery of Evolved Car Controlling Programs

```
(Method:rootGlue
   (Method:sub
      (Method:getSpeed
      )
      (Method:sub
         Constant : 3.0
         Constant : 4.0
      )
   )
   (Method:mul
      (Method:angleToNextWaypoint
      )
      Constant : 0.2617993877991494
   ))
```

Figure 4: Sample evolved program using standard GP

This section presents some sample evolved car controlling programs. A very simple strategy that seems to be adequate of achieving a high fitness on track 5 was based on maintain-

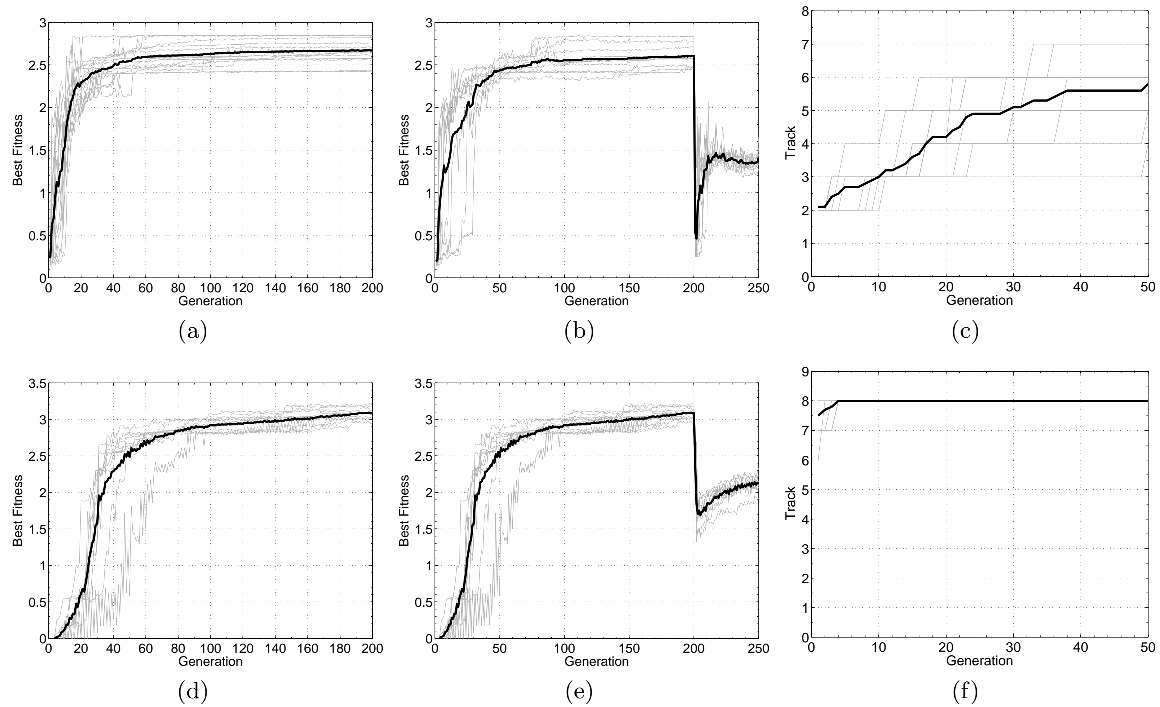

Figure 3: (a) Best-of-generation individuals using GP with ADFs; (b) Best-of-generation individuals using OOGP and MM (generalization to additional tracks is shown after generation 200); (c) Tracks driven proficiently using OOGP and MM; (d) Best-of-generation-individuals using MLP; (e) Generalization to additional tracks after generation 200 using MLP (f) Tracks driven proficiently using MLP

```
(Method:rootGlue
   Constant : 3.141592653589793
   (Method:sub
      (Method:getWallSensorReading
         Constant : -1.0
         Constant : 0.2617993877991494
      )
      (Method:angleToNextWaypoint
      )
   ))
```

Figure 5: Sample evolved program using standard GP

ing a constant speed in the forward direction and steering defined as a factor of the angle to the next waypoint. Various bloated forms of this algorithm have been designated as the best-of-run controller, using all different forms of program representation (OO/Functional with ADFs/Functional with no ADFs). Figure 4 presents a program that exhibits the above strategy. The `rootGlue` accepts two sub-expressions rooted is the subtraction (sub) and multiplication (mul) methods respectively. The evaluation of the first subtree adds one to the current speed. The evaluation of the second subtree multiplies the angle to next waypoint with the coefficient 0.26. More specifically the above controller attained a fitness of 2.84 in track 5 but generalized poorly when tested on additional tracks.

Figure 5 presents a controller that generalised with the first 6 tracks. Here we observe a different strategy than the one previously discussed. First we note that the controller is constantly driving forwards, given by the constant 3.14. The tree that is evaluated to induce the steering value uses two sensors. The wall sensor can be visualized as a line protruding to the right of the (with an angle of approximately $\pi/3$) and a range of approximate pixels. The `wallSensorReading` method returns a value in the interval of [0, 1]. Given that the range of the sensor is relatively small (50 pixels), this sensor will be most of the time returning 1 unless it comes very close to a right wall (assume that the can is moving anti-clockwise). The car will initially turn away from the first way point. At the initial time-step

```
(Method:rootGlue
     Constant : 2.0
     (Method:sub
        Constant : 0.2617993877991494
        (Method:sub
           (Method:setValue
              SettableVariable[1]
              (Method:angleToNextWaypoint
              )
           )
           (Method:mul
              Constant : -0.5235987755982988
              Constant : 0.0
           )
        )))
```

Figure 6: Sample evolved program evolved using OOGP

t_0 the angle to next way point is zero so the expression tree rooted in the `sub` node will return a negative value. While the car is steering to the opposite direction, the angle to the next way point will be increased and thus the difference $1 - angleToNextWayPoint$ will gradually return a negative number that will force the car to change direction and move towards the way point.

Finally, figure 6 presents a program that uses state information. The controller is driving forwards as specified by the constant 2. The steering value is given by the subtree rooted in the node `sub`. At time t_i the program stores the angle to the next way point to the instance variable with index 1 and returns the value of the instance variable at time t_{i-1}.

4. GP RESULTS

Observing the results of table 2 we note that state information did not add any significant advantage to the evolved controllers and so the functional representation that employs ADFs performed quite similar to the OO representation. As expected, GP with ADFs performed better than standard GP. Furthermore, we found that even when state manipulation constructs are present in the primitive language, the evolved programs presented a tendency towards avoiding their use. Our intuition suggests that this may be due to the nature of the problem. The evolved programs can have direct access to the information needed for navigation within the track and it seems that there is no need for the program to maintain an environmental model by keeping track of various sensor readings. State manipulation would appear very promising if we allow the car to be trained in the normal way and then we switch off the environmental sensors. It seems likely that a sensorless program could take advantage of extra information that it built and maintained during the training stage. However, regardless of the fact that the solution to this problem did not require the involvement of state variables, their addition to the primitive language along with the setter method did not increase the required effort to induce proficient controllers.

Homologous uniform crossover appears to be very promising and achieved substantially competitive performance with macromutation as far as the training on a single track in concerned. However, controllers evolved using recombination found to be generalising with an average of 4.6 tracks whereas those evolved using macromutation drove sensibly in an average of 5.8 tracks. This form of crossover is very sensitive to the evolutionary run parameters (initial depth, population size, point mutation probability) and is believed that adequate tuning will yield even better performance.

5. DISCUSSION AND CONCLUSIONS

The main finding of our experiments is that, for the given problem and experimental setup, the various versions of GP evolve faster than neural networks, but the neural networks ultimately perform better, especially on the more complicated version of the task, that takes all eight tracks into consideration.

However, we don't understand why this is the case. An early suspicion was that, as the GP trees evidently make use of much fewer wall sensor readings than the neural network controllers, this contributed to the fast learning but poor generalisation of GP controllers. The argument was that it is easier to learn to drive on a simple track when only caring about the speed and the angle to the next waypoint, but that this strategy breaks down when exposed to the more complicated tracks where the straight line between two waypoints might pass through a wall. As neural networks are in effect forced to consider all its sensor readings, this makes it harder to find an initial control strategy, but once found, such a control strategy will be much more robust. Much to our dismay, our hypothesis was falsified by the inclusions of the "minimal" neural network controller that only uses four wall sensors, yet performs almost as well as those controllers that use 12 wall sensors.

Another hypothesis is that we are being unfair to GP when we are asking it to compete with neural networks on the neural networks' home arena. GP experiments typically use much larger population sizes and different selection and crossover regimes. Given such changes the GP controllers might very well outperform our neural network controllers.

Another puzzling phenomenon is the virtual lack of difference between the performance of stateless and stateful controller representation. Our best bet as to why this is so is that we need even more complex versions of the car racing task, such as competitive multi-car racing, in order to exploit the statefulness of the controllers. Maybe we also need to introduce more complex primitive object; one promising suggestion is to include complete forward models of the car dynamics as objects which could be accessed and manipulated by the OOGP system.

6. REFERENCES

[1] A. Agapitos and S. M. Lucas. Evolving a statistics class using object oriented evolutionary programming. In *Proceedings of the 10th European Conference on Genetic Programming*, 2007.

[2] E. Kirshenbaum. Genetic programming with statically scoped local variables. Technical Report HPL-2000-106, Hewlett Packard Laboratories, Palo Alto, 11 Aug. 2000.

[3] J. Koza. *Genetic Programming II: automatic discovery of reusable programs.* MIT Press, Cambridge, MA, (1994).

[4] J. R. Koza, D. Andre, F. H. Bennett III, and M. Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufman.

[5] W. B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming.* Kluwer, Boston, 24 Apr. 1998.

[6] R. Poli and W. B. Langdon. Genetic programming with one-point crossover and point mutation. Technical Report CSRP-97-13, University of Birmingham, School of Computer Science, Birmingham, B15 2TT, UK, 15 Apr. 1997.

[7] A. Teller. The evolution of mental models. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

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

[9] J. Togelius and S. M. Lucas. Arms races and car races. In *Proceedings of Parallel Problem Solving from*

Nature. Springer, 2006.
[10] J. Togelius and S. M. Lucas. Evolving robust and specialized car racing skills. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.