Multiobjective techniques for the Use of State in Genetic Programming applied to Simulated Car Racing

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

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

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

***Abstract*— Multi-objective optimisation is applied to encourage the effective use of state variables in car controlling programs evolved using Genetic Programming. Three different metrics for evaluating the use of state within a program are introduced. Comparisons are performed among multi- and single-objective fitness functions with respect to learning speed and final fitness of evolved individuals, and attempts are made at understanding whether there is a trade-off between good performance and stateful controllers in this problem domain.**

I. Introduction

An essential component of intelligent behavior is the ability to extract, store and utilize information about the environment. Maintaining a mental environmental model may allow an agent to plan its actions more effectively by combining immediate sensory information along with 'memories' that have been acquired while operating on that environment. Puzzlingly, evolutionary learning techniques in general and Genetic Programming (GP) in particular do not typically utilize this particular type of learning.

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 maintenance and 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, simply because many problems can be solved without the use of state. GP is known for exploiting loopholes and thus to realistically encourage the evolution of effective storage/retrieval of intermediate state information previous experimenters have been handcrafting the fitness environment in such a way as to provide an evolutionary selection pressure towards the use of memory.

In [1] we investigated the evolution of controllers with internal state for a version of a simulated car racing problem, using both GP and neuroevolution. The conclusions were somewhat disappointing, as we found no significant differences between the final fitnesses of controllers evolved using representations that allowed for the evolution of stateful controllers (recurrent networks and GP with state variables), and those that restricted the evolved controllers to be reactive. Further, analysis of the evolved GP trees showed a tendency towards avoiding the use of primitives for setting and accessing state information.

One possible reason for this result is that the version of the single car racing problem we were investigating was too simple. We know from earlier investigations that adding another car makes the car racing problem much more challenging, to the point where controller representations which can easily evolve human-competitive results on single-car versions of the problem, fail to come anywhere near human-competitiveness on two-car versions of the problem [2]. To demonstrate the complexity of this version of the problem, a controller would need to model the trajectory of the competing car, and ultimately the behaviour of the competing driver, in order to win against a really good competitor.

However, it could also be that the problem of single car racing in itself is complex enough, but there is something else about the evolutionary process that prevents the emergence of stateful controllers. Our intuition suggests that there was not adequate selection pressure so as to guide the evolving population of programs towards exploiting the use of state (see [1] for a discussion) and thus purely reactive controllers were evolved. In this paper, we introduce simple metrics for the efficient use of state in a program, make them explicit

objectives for evolution, and use multiobjective optimisation to understand whether there is a trade-off between good performance and stateful controllers.

II. 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 (ER), 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 [3], how to generalise controllers to reliable drive on several dissimilar tracks and specialise them for particular tracks [4] and the impact of fitness functions on competitive co-evolution of two cars on the same track [2].

Puzzlingly, very little work in evolutionary robotics seem to make use of the vast knowledge accumulated in the sister field of 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.

III. State in Genetic Programming

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 [5]. 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 generalised both the above to the notion of an *Automatically Defined Store* [6]. Finally, Kirshenbaum (2000) presented work on the evolution of programs that use statically scoped local variables [7]. That is, variables whose visibility is bounded to a given scope defined by a subtree rooted on a `Let` construct.

While knowledge and experience of using state variables in GP was being accumulated, researchers started applying these ideas to solve interesting problems, and ultimately introduce a way of making the GP paradigm *Turing Complete* (though this required the complementary use of iteration or recursion constructs). Realistically, it is not possible in a paper of this length to review all previous attempts at evolving stateful programs and we will restrict our discussion in those studies that employed an expression-tree representation of the evolvable individuals. The motivation for doing this stems from the need to understand how previous experimenters have been crafting the fitness environment so as to encourage the use of state.

Teller [5] evolved programs that solve the problem of pushing blocks up against the boundaries of a world represented as a toroidal grid. He used a very interesting strategy to necessitate the use of memory by strictly limiting the function sets so that the evolved programs could move only once per evaluation and received very limited sensory feedback. Without using state only limited fitness was possible. Andre [8] tackled the problem of an agent whose task is to collect all of the gold scattered in a five-by-five toroidal grid. To encourage the use of memory representation, the evaluation of an individual occurred in two stages, namely, map-making and map-using. In the first stage, the agent was allowed to move around the world and write to a five-by-five memory, but not pick up any gold. In the second stage, the agent can access its memory, but is unable to see the gold. Brave [9] studied a similar problem of an agent that explores the world and is required to produce a plan for reaching every arbitrary location in the world from every arbitrary starting point. He used a dual-phase fitness function similar to that used in Andre's experiments. Langdon [10] and Bruce [11] independently evolved Abstract Data Types (ADTs) such as stacks, priority queues and linked lists. They investigated the difficulty of evolving methods that needed to cooperate such that some shared memory would be used in a compatible way. For example, when implementing a stack it is vital that the push and pop methods cooperate properly. For this purpose the fitness assignment of programs is performed in a two-pass process where the evaluation of methods that modify memory precedes the evaluation of methods that query it. We have previously applied *Object Oriented Evolutionary Programming* (OOEP) [15] to the task of evolving complete classes that implement an interface of methods that perform statistics on a sample of data. In order to encourage the use of state variables we employed a similar multi-phasic fitness assignment process in which input data is first added to the statistical sample to modify the internal state of the `Statistics` object and hence methods for computing the mean, variance and standard deviation are evaluated based on that state.

However, a particularly interesting study is the one re-

ported by Spector and Luke [12] on the exploitation of cultural information. They implemented culture by having all individuals to share the same memory which is initialised only at a start of a genetic programming run. In this way, a program may pass information to itself, to its contemporaries, to its offspring, and to unrelated members of future generations. One would expect that the nature of the symbolic regression problem being tackled and the typical training process employed would provide no selection pressure towards programs that are consulted by this pool of cultural information. Nevertheless, results indicated that GP combined with culture decreased the computational effort required to induce target solutions when compared to GP variants that use no memory or standard *indexed memory*.

This literature review suggests that traditionally the way of encouraging the evolution of stateful individuals has been performed by limiting the sensory input and/or devising a fitness assignment process that explicitly sets the order in which a program's modules are being evaluated. One thing that has become apparent is that there should be adequate selection pressure to evolve programs that use memory.

IV. Measuring the Use of State Variables in a Program

As discussed in section I it seems unlikely that the inclusion of language primitives for modifying and inspecting a state space would prove to be advantageous for an arbitrary problem. Solutions using memory may be less complex than those not using memory, but may be harder to evolve. 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 modules or even within the same module) 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.

We devised a set of metrics for measuring the effective use of state variables within a program. One particular observation made by analysing the evolved expression-trees in [1] is that state manipulation and inspection primitives were hardly used in the best-of-run individuals. We monitored the evolution of use of such primitives in each generation simply by counting their instances within the population expression-trees. We noticed that throughout the evolutionary runs their number decreased as generations elapsed. Clearly, on a structural level, if we wish to evolve programs that use state information we initially need to define a way of preserving their presence in the population genetic pool.

The first metric is defined as the ratio of the number of variables used within a program to the number of variables offered for use by the primitive language. That is,

$$Metric_1 = \frac{variablesUsed}{variablesOffered} \tag{1}$$

A substantial issue about the use of state information within a program is the interplay between setting and accessing the value of state variables. Typically, in a human-written program one would expect the value of a variable to be set prior its use. Ultimately, it seems at least bizarre for a program to access state variables that have not been initialised (most compilers will not accept this) or set to a particular value during the computation. We wish to encourage the proper use of state by defining the ratio of the number of variables being set within the program to the number of variables being accessed. To calculate this, we first need to trace the primitives that set specific state variables and hence map these to primitives that access the very same state variables. This ratio is presented bellow.

$$Metric_2 = \frac{variablesSet}{variablesAccessed} \tag{2}$$

In the case where the number of primitives that access state variables is zero we arbitrarily choose to divide by 10^{-5}.

Previous studies have already identified one major troubling aspect of the use of global state variables in GP. Spector [13] terms this as *data dependencies* whereas Woodward [14] as *global memory interference* and they both refer to the problem of having functions and terminals to write to a shared memory mechanism, or take turns manipulating a global environment in an explicit order. The application of variation operators can have a concomitant impact (by destroying these dependencies) on the operations of other functions and terminals not only in a particular subtree but throughout an individual. One possible way to counteract the destructive effects of variation operators, as these are experienced by destroying dependencies between *setting* and *getting* primitives, would be to request that some form of distance between these primitives within the expression-tree is kept at minimum. Intuitively, this could encourage the formation of fine-grained blocks of code that perform memory manipulation and limit their destruction via the variation operators. For this we need to calculate the *positional distance* of *setter-getter* pairs within the individual and average this over the number of these pairs. This metric is defined below:

$$Metric_3 = \frac{\sum_{i=1}^{n} Distance(SetGetPair)}{n} \tag{3}$$

where $Distance(SetGetPair)$ is the positional distance within the expression-tree of two primitives that set and access a particular state variable and n is the number of these pairs.

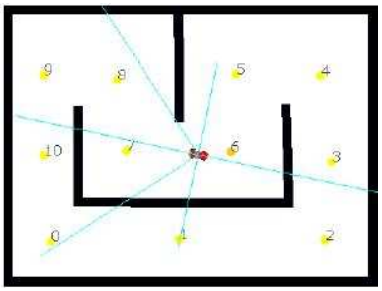

Fig. 1. The track used in the experiments

V. 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 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 a track it manages to travel in 500 time steps. This is measured by how many way points it manages to pass within that time. The racing track is defined by a starting position, a set of walls and a chain of way points. The track used for the experiments in this paper is depicted in figure 1. The lines protruding from the car are the visualizations of wall sensors. In this particular example the car is using 6 sensors whose angles and ranges are specified by respective parameters (see later section). 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 500 time steps yields fitness of 1, passing fewer way points leads to lower fitness and completing several laps of the track within the allotted time gives higher values.

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

Fig. 2. The interface specifying the signature of the driving method

VI. Object-Oriented Genetic Programming

A. Evolvable Class Representation

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 2 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. Following our previous work on the evolution of complete classes [15] 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.

B. State Representation

We employed a simple memory addressing scheme by combining *type* information along with the mechanism of *pass by reference* [10] 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.

C. Variation Operators

Our search employs a mutation-based variation scheme. For our purposes here, *subtree macro-mutation* (MM) is applied by substituting a node in the tree with an entirely randomly generated subtree of the same return type, under depth or size contraints). 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, 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.

D. 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 I. 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 `doubles` 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 2). 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(`$\pi/2$`, 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.

E. Evolving Object-Oriented Controllers

The genome representation in the case where we evolve a complete class has been outlined in section VI-A. 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 as described in [16] along with the addition of randomly selecting the number and type of instance variables. The number of additional instance methods (ADFs) is set to the random interval of [1, 3] 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 [16], 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. Additionally, 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 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 time-step 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 500 time-steps, and finally dies along with the cease of fitness evaluation procedure.

VII. Methods

A. Experimental Context

We wish to explore the idea of using multi-objective optimisation to encourage the effective use of state variables within the evolvable individuals. However, our primary concern is to investigate whether there is a trade-off between good performance and a stateful controller. Our intuition suggests that if the evolutionary process finds some workable representation of the useful environmental features, it may

TABLE I

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
Parameters	subject to environment model of evaluation	double

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. Other than simply considering the number of waypoints passed in order to reward an individual, it is hoped that the various metrics defined in section IV will provide a selection pressure towards individuals that use state variables ($Metric_1$) and most importantly will encourage the interplay between setting and accessing state information to be performed in a compatible way ($Metric_2$). Finally, we also wish to reward programs which 'protect' the dependency among *setter* and *getter* primitives by keeping their positional distance minimised.

In our previous paper [1] we observed that most of best-of-run individuals where exploiting the use of the sensor that returns the angle to the next waypoint in order to determine the steering direction. This very simple strategy proved to be sufficient of obtaining high fitness. Here, we deliberately avoid the inclusion of such construct in the primitive language. We defined four different families of environmental sensors and used each one separately to populate the method set of a particular evolutionary run. These sets are the folowing: S_1 = {wallSensorReading (WSR)}, S_2 = {wallSensorReading (WSR), distanceToNextWayPoint (DNWP)}, S_3 = {wallSensorReading (WSR), speed (SP), distanceToNextWayPoint (DNWP)}, and S_4 = {wallSensorReading (WSR), speed (SP)}. For the shake of comparison we run the same experiments with two different GP variants. A first one which allows the use of state variables but employs a single-objective fitness function (based on number of way points passed) and a second one which employs the same fitness function but does not allow for the involvement of state variables in the primitive language - a stateless individual (see [1] for details). Finally, each experiment is performed 10 times.

B. Evolutionary Algorithms

For single-objective optimisation we used a panmictic, generational genetic algorithm (GA) combined with elitism (1%). The algorithm uses tournament selection with a tournament size of 2. In this case, the scalar fitness of an individual is defined simply as the number of waypoints passed within 500 time steps. For multi-objective optimisation we employ the *Non-Dominated Sorting Genetic Algorithm II* (NSGA-II) as this is presented in Deb (uses binary tournament selection). The objective vector consists of 4 objectives, namely: (a) the number of way points passed within 500 time steps (to be maximised), (b) the ratio of variables used to variables offered ($Metric_1$ - to be maximised), (c) the ratio of variables set to variables accessed ($Metric_2$ - to be maximised), and (d) the average positional distance of *setter-getter* primitives ($Metric_3$ - to be minimised). To alleviate the considerable noisiness of the fitness evaluations, the number of way points passed is calculated as the average of three independent trials in both evolutionary algorithms.

C. GP Run Parameters

The population size was set to 100 and evolution proceeds for 100 generations. All initial populations are created using the *Ramped-Half-and-Half* algorithm with a maximum initial depth of 5. Subsequently, expression trees are allowed to grow up to depth of 10. Here, macro-mutation (MM) is applied with 100% probability.

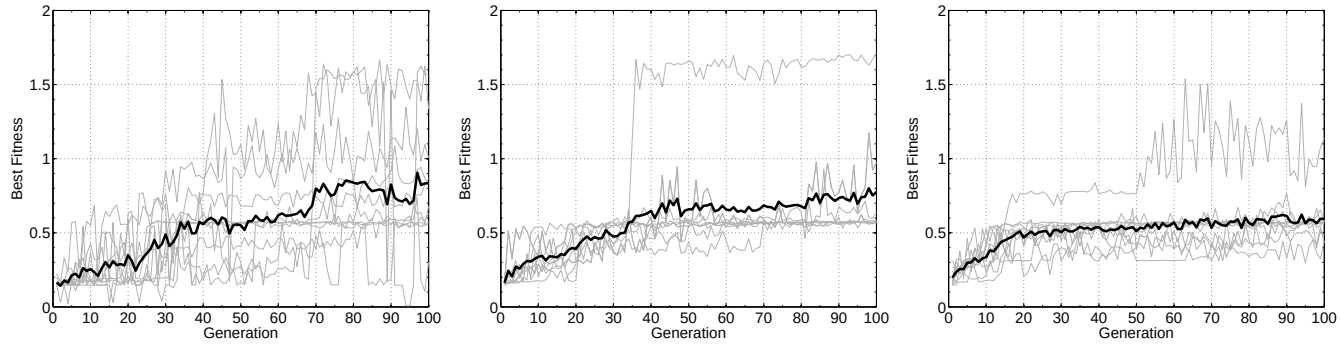

Fig. 3. Best-of-generation individuals using wall sensor reading and distance to next way point as sensory input: (a) multi-objective, (b) single-objective (c) single-objective with no state variables

VIII. Results and Discussion

Table II presents the evolution of best-of generation fitness (as this is measured by the number of waypoints passed) averaged over 10 independent evolutionary runs. The first observation on the results is that GP runs that used multi-objective optimisation to encourage the use of state variables performed slightly better than those that allowed state variables but used single-objective optimisation solely based on the number of way points passed. Ultimately, those runs that did not allow state variables performed considerably poor. This justifies our initial hypothesis about not including the angle-to-next-way-point sensor in the expectation of making the task more challenging. It seems that the effective use of state variables is necessary to achieve better performance.

Figures 3(a), 3(b), 3(c) illustrate the evolution of best-of-generation individuals using (a) multi-objective optimisation for stateful controllers, (b) single-objective optimisation for stateful controllers, and (c) single-objective optimisation for stateless ones. It can be seen that regardless of the fact that the average fitness of (a) is slightly better than that of (b), individuals evolved using a Pareto-based fitness function that considers the evaluation of state use deviate more from that average as opposed to the ones evolved using a scalar fitness function - a result that is substantially encouraging. It is believed that we may be unfair to the NSGA-II algorithm when we are asking it to compete with a single-objective GA under the current population size and number of generations. This can be backed-up by the observation made in regards to learning speed as this is depicted in figures 3(a) and 3(b). We note that the single-objective GA (fig. 3(b)) learns slightly faster than NSGA-II but it seems to be stagnating at around generation 50 (after gen. 50 there exists only a slight improvement). On the other hand NSGA-II seems to be learning more smoothly up until the last generation. Given an increase on the number of generations, one would expect that a multi-criterion fitness function might very well outperform the one based solely on the number of way points passed.

Finally, we note that there is no significant performance difference among the experiments that use different families of sensory inputs. However, it is worth noting that all of these sets of primitive elements include the wall-sensor-reading (WSR). An early suspicion is that the GP trees evidently make use of much fewer sensor readings than those supplied in the method set. The use of wall-sensor-reading is necessary to decide upon the steering direction at any point in time and is believed that this sensory input can be adequate to achieve good driving performance even in the case where the car is always driving forward at a constant speed. This is in line with the results reported in [1] where we observed that most of best-evolved-controllers presented a tendency towards avoiding the use of most sensors. It was hard to reason with the evolved controllers as their convoluted nature was opaque to human understanding (given also the *lazy evaluation* of the `If-Then-Else` construct). A static analysis in terms of the primitives residing in the evolved individuals is currently under way.

IX. Conclusions and Future Work

In this paper we attempted to use multi-objective optimisation to encourage the emergence of stateful car controlling programs. Three different metrics for evaluating the use of state within a program were introduced and fed as objectives in a Pareto-based fitness function used by NSGA-II along with the ultimate objective of racing performance (as defined by the number of way points passed). By optimising the objectives defined by the state-use-metrics we expected to provide an evolutionary selection pressure towards the effective use of state variables both in terms of compatibility between *setting* and *getting* a specific state variable and also in terms of structure by requesting that *setting* and *getting* primitives lie close to each other. The current simulated car racing environment, as this is perceived by the sensory inputs, necessitates the use of state information. In this environment we performed a comparison between the evolution of programs that are allowed state variables but are rewarded only for racing performance and those programs that allow state variables but are also rewarded for the way in which they utilise this state information. Results were encouraging, and the optimization of both racing performance and state-variables-use (using NSGA-II) turned out to achieve better

TABLE II

AVERAGE FITNESS OF BEST CONTROLLER FOR GENERATIONS 10, 50, 75, 100 (AVERAGED OVER 10 INDEPENDENT EVOLUTIONARY RUNS - STD. DEVIATION IN PARENTHESES)

Method	**10**	**50**	**75**	**100**
Multiobjective/WSR	0.26 (0.11)	0.37 (0.21)	0.5 (0.2)	0.8 (0.32)
Multi-objective/WSR, DNWP	0.25 (0.14)	0.55 (0.28)	0.76 (0.43)	0.83 (0.38)
Multi-objective/WSR, SP, DNWP	0.21 (0.11)	0.39 (0.24)	0.41 (0.29)	0.68 (0.22)
Multi-objective/WSR, SP	0.22 (0.15)	0.59 (0.24)	0.75 (0.37)	0.78 (0.27)
Single-objective/WSR	0.4 (0.14)	0.56 (0.26)	0.7 (0.36)	0.74 (0.35)
Single-objective/WSR, DNWP	0.33 (0.11)	0.65 (0.35)	0.68 (0.34)	0.77 (0.36)
Single-objective/WSR, SP, DNWP	0.31 (0.11)	0.61 (0.14)	0.67 (0.18)	0.76 (0.29)
Single-objective/WSR, SP	0.29 (0.1)	0.6 (0.31)	0.71 (0.47)	0.75 (0.46)
Single-objective (no state vars)/WSR	0.44 (0.07)	0.62 (0.25)	0.64 (0.23)	0.65 (0.23)
Single-objective (no state vars)/WSR, DNWP	0.33 (0.11)	0.51 (0.13)	0.57 (0.29)	0.59 (0.2)
Single-objective (no state vars)/WSR, SP, DNWP	0.42 (0.08)	0.54 (0.08)	0.56 (0.08)	0.59 (0.14)
Single-objective (no state vars)/WSR, SP	0.37 (0.12)	0.56 (0.17)	0.65 (0.15)	0.67 (0.29)

performance.

This investigation can hardly be said to be without practical interest even outside of this evolutionary robotics application. Searched-based Software Engineering is a newly coined term and is concerned with embodying all knowledge accumulated about designing and implementing complex software artifacts into evolvable software systems. State is an absolute requirement for certain computations and its effective use by the GP paradigm could yield great benefits. However, more applications of our methodology need to take place in order to evaluate its breadth of efficiency.

For future work, we plan to investigate the use of multiobjective optimisation for evolving populations of behaviourally different controllers. One of the motivations for this is that we want to map the space of viable behaviours on the current problem - how many different ways are there of achieving good driving skills? Another motivation is to explore a potential method of generating sets of interesting opponents for computer games. In many games, the challenge for artificial intelligence is not so much to make the computer-controlled agents behave as well as possible (as in driving as fast as possible, or killing the player as quickly as possible) but to make the agents behave in an interesting way. The way multiobjective optimisation could help us with this is by evolving pareto fronts of agents in several behavioural dimensions, letting us automatically pick agents that perform especially well in some dimension and at least decently well in others. For example, we could have car drivers that driver very fast on their own but are bad at overtaking cars and blocking other cars from overtaking them, and other drivers that are experts on out-maneuvering competitors in close interactions but less good at taking corners. Of course, this requires that we can measure relevant aspects of the behaviour of the car, an issue which is currently under investigation.

REFERENCES

[1] A. Agapitos, J. Togelius, and S. M. Lucas, "Evolving controllers for simulated car racing using genetic programming," in *Proceedings of the Genetic and Evolutionay Computation Conference*, 2007.

[2] J. Togelius and S. M. Lucas, "Arms races and car races," in *Proceedings of Parallel Problem Solving from Nature*. Springer, 2006.

[3] ——, "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] A. Teller, "The evolution of mental models," in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, ch. 9, pp. 199–219. [Online]. Available: http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/MentalModels.ps

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

[7] E. Kirshenbaum, "Genetic programming with statically scoped local variables," Hewlett Packard Laboratories, Palo Alto, Tech. Rep. HPL-2000-106, 11 Aug. 2000. [Online]. Available: http://www.hpl.hp.com/techreports/2000/HPL-2000-106.pdf

[8] D. Andre, "Evolution of mapmaking ability: Strategies for the evolution of learning, planning, and memory using genetic programming," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, vol. 1. Orlando, Florida, USA: IEEE Press, 27-29 June 1994, pp. 250–255.

[9] S. Brave, "The evolution of memory and mental models using genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Stanford University, CA, USA: MIT Press, 28–31 July 1996, pp. 261–266.

[10] W. B. Langdon, *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, ser. Genetic Programming. Boston: Kluwer, 24 Apr. 1998, vol. 1. [Online]. Available: http://www.wkap.nl/prod/b/0-7923-8135-1

[11] W. S. Bruce, "Automatic generation of object-oriented programs using genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*.

[12] L. Spector and S. Luke, "Cultural transmission of information in genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Stanford University, CA, USA: MIT Press, 28–31 July 1996, pp. 209–214.

[13] S. Luke and L. Spector, "A comparison of crossover and mutation in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds. Stanford University, CA, USA: Morgan Kaufmann, 13-16 July 1997, pp. 240–248.

[14] J. Woodward, "Evolving turing complete representations," in *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, Eds. Canberra: IEEE Press, 8-12 Dec. 2003, pp. 830–837.

[15] 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.

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