Research Repository UCD

Title	Learning environment models in car racing using stateful genetic programming
Authors(s)	Agapitos, Alexandros, O'Neill, Michael, Brabazon, Anthony, Theodoridis, Theodoros
Publication date	2011-08-31
Publication information	Agapitos, Alexandros, Michael O’Neill, Anthony Brabazon, and Theodoros Theodoridis. “Learning Environment Models in Car Racing Using Stateful Genetic Programming.” IEEE, August 31, 2011. https://doi.org/10.1109/CIG.2011.6032010.
Conference details	Paper presented at the 2011 IEEE Conference on Computational Intelligence and Games (CIG’11), Seoul, South Korea, Aug.31-Sept.3, 2011
Publisher	IEEE
Item record/more information	http://hdl.handle.net/10197/3573
Publisher's statement	2011 IEEE
Publisher's version (DOI)	10.1109/CIG.2011.6032010

Downloaded 2026-05-01 23:44:06

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

© Some rights reserved. For more information

Learning Environment Models in Car Racing using Stateful Genetic Programming

Alexandros Agapitos, Michael O'Neill, Anthony Brabazon, Theodoros Theodoridis

Alexandros Agapitos, Michael O'Neill, and Anthony Brabazon are with the
Financial Mathematics and Computation Research Cluster,
Natural Computing Research and Applications Group,
Complex and Adaptive Systems Laboratory,
University College Dublin, Ireland.

Theodoros Theodoridis is with the School of Computer Science and
Electronic Engineering, Dept. of Robotics, University of Essex, Colchester, UK.

Abstract

For computational intelligence to be useful in creating game agent AI we need to focus on methods that allow the creation and maintenance of models for the environment, which the artificial agents inhabit. Maintaining a model allows an agent to plan its actions more effectively by combining immediate sensory information along with a memories that have been acquired while operating in that environment. To this end, we propose a way to build environment models for non-player characters in car racing games using stateful Genetic Programming. A method is presented, where general-purpose 2-dimensional data-structures are used to build a model of the racing track. Results demonstrate that model-building behaviour can be cooperatively coevolved with car-controlling behaviour in modular programs that make use of these models in order to navigate successfully around a racing track.

1 Introduction

An essential component of intelligent behaviour is the ability to extract, store and utilise information about the environment. Maintaining an environment 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 in that environment [1]. It is very likely that most interesting problems do not have solutions that are simple mappings from inputs to correct outputs; some kind of internal state or memory is needed to operate optimally in these domains. Surprisingly, evolutionary learning techniques in general and Genetic Programming (GP) [2] in particular, do not typically utilise this particular type of learning.

Most GP-evolved programs evolved so far are simple expression-trees that perform simple mappings from inputs to desired outputs, and typically do not utilise state variables nor employ side-effecting primitive constructs. The term stateful GP (or GP with memory [1]) encompasses those GP program representations that allow some sort of program state to be maintained either throughout a single program execution (i.e. state takes the form of a local data-structure), or in-between numerous consecutive program executions (i.e. state takes the form of a non-transient global data-structure). The task of learning how to use memory by means of stateful GP requires an evolutionary search in the space of imperative programs. In computer science, imperative programming is a programming paradigm that expresses computations in terms of a program state and statements that alter the program state. From a low-level perspective, the program state is defined by the contents of memory registers. Assignment statements, which sit at the heart of imperative programming, perform an operation on information located in memory and store the results in memory for later use. 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, and further to object orientation. A comprehensive literature review on the evolution of stateful programs by mans of GP can be found in [3]; it can be safely stated that this revealed a very limited number of attempts in this research area.

This paper tackles the problem of environment modelling for non-player characters in computer games using stateful GP. We apply this technique to the open-source car racing computer game by the name of TORCS (`http://torcs.sourceforge.net`). The current work is an initial investigation of the application of stateful GP to TORCS, and explores the feasibility of:

1. Learning a racing track model, represented as an instantiation of a general-purpose data-structure.
2. Using this model to navigate around the track.

We are currently not interested in learning how to navigate around the track as fast as possible, as we felt that this is a challenging research area on its own right, and may compose a hindering factor by introducing complexity to our modelling technique. In this part of the work, the criterion for success is the emergence of model-building ability, thus we do not provide a comparison on the racing speed against GP controllers that do not utilise memory. It is an absolute requirement, however, that further research into this area should seek to compare the performance against other benchmark drivers. In this work, we concentrate on building track models in computer memory, and use those to successfully navigate around the track. In order to ensure that the evolved drivers utilise the model for navigation, they are required to race using a minimum of sensory information, limited to the car's current velocity, the car's distance from the start line, and the car's distance from the track axis.

The contribution of this paper is twofold. First, in the area of game agent AI, it provides a method to evolve agents that use state effectively to build models of the environment given sensory information in an initial model-building process, and then utilise this model to effectively navigate and plan. Our current focus is to study a

model-building method based on GP with *indexed memory* [1], applied to the car racing domain. The investigation of building racing track models in computer memory is closely mirrored by the ongoing investigation of how to best evolve programs with state in GP. We believe there can be much fruitful interplay between these two research domains. The second contribution is to further evaluate indexed memory as a general tool for using memory in GP. We expect that for particular classes of problem, such program representations provide a more appropriate unit of evolution than the traditional, essentially stateless approach to expressing computations.

The rest of the paper is organised as follows. Section II outlines previous work on the evolution of car racing controllers, work on the automatic induction of environment models via stateful GP, as well as previous approaches to track model-building in car racing games. Section III collectively presents the method for learning environment models. Firstly, it presents the multi-phasic fitness evaluation approach to the cooperative coevolution of program functionality which performs model-building, and track navigation. Secondly, it describes the stateful program representation language, and the details of the evolutionary algorithm employed. Section IV presents the experimental results, while Section V draws our conclusions and sketches future research directions.

2 Previous Work

2.1 Evolving Car Racing Controllers

The evolution of car racing controllers has already attracted attention from the ML community. The majority of controllers in evolutionary car racing have been represented as feed-forward multilayer perceptrons, recurrent neural networks, and various forms of GP. Togelius et al., in their own version of a car racing simulator, have previously investigated how to best evolve controllers for single-car, single-track racing [4], how to generalise controllers to reliably drive on several dissimilar tracks and specialise them for particular tracks [5], and the impact of fitness functions on competitive co-evolution of two cars on the same track [6]. The controllers in those experiments employed a neural network representation. Furthermore, in [7], Togelius et al. investigated various flavours of coevolutionary learning, and different controller architectures to evolve car racers. There, they contrasted between neural network and GP representations.

In the domain of GP, Agapitos et al. studied the performance of various program representations for evolving car controllers [8]. They contrasted the performance of stateless versus stateful representations, as well as as representations that allow for modules versus those that are based on a single expression-tree architecture. Their subsequent work in the same area reports experimental results that employ multi-objective optimisation to encourage the effective use of state variables in the evolution of car racing controllers [9]. Additionally, in [10], Agapitos et al. presented experiments, where a number of partially conflicting objectives were defined for car racing game competitors, and multi-objective evolution of GP-based controllers yielded Pareto fronts of interesting and behavioural diverse driver opponents. The most recent work that employed a GP-based representation is reported by Ebner and Tiede [11], in which car

racing controllers represented by multi-tree program architectures have been evolved out of simple arithmetic and sensory-input primitive constructs.

A review of previous work on the application of GP to TORCS suggests that evolved car racing controllers do not utilise any sort of learning of an explicit model of the environment. Controllers are typically purely reactive, utilising a variety of ego-centric (i.e. first-person) sensors whose values form the basis of the output control commands.

2.2 Evolving Environment Models with Stateful GP

The current work is based on advances in the evolution of environment models in artificial-life agent-based simulations by means of stateful GP. Teller [12] studied the evolution of agents that build and maintain mental models. For that purpose he designed Tartarus, which is an NxN toroidal grid-world. The agent's goal is to push all the boxes out of the center of the grid onto the perimeter grid positions. Programs were evolved that solved the problem of pushing blocks up against the boundaries of the world. He biased the evolutionary search towards programs that utilise state variables 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 [13] 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 in the evolved programs, 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. Andre found that this approach can evolve programs that store simple representations of their environments, and use these representations to produce simple plans of actions. Finally, Brave [14] 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. Brave's work corroborated previous results on the validity of indexed memory as a powerful extension to GP systems that learn how to utilise memory structures, and found that the path-planning solution could be synthesised by accessing a memory store whose structure is not defined a priori but evolves to create a directed graph.

2.3 Building a Model of the Racing Track

In the domain of car racing games, learning a model of the racing track has the potential of integrating information into a planning process, thus allowing a car racing controller to be introspective rather than purely reactive. As a concept, it has already been explored in the domain of TORCS. The work of [15] introduced the idea of recording crash points to improve the performance from previous experience. Of direct relevance to our work is the previous work of [16] on generating track models. The technique was based on estimating the curvature of the next invisible part of the track in order to classify track segments accordingly. The classification was then used to adjust the

speed of the car. A mechanism for aggregating this information allowed the formation of complete track models.

3 Evolutionary TORCS

3.1 Cooperative Coevolution of Model-builder and Car-controller Branches

Our approach to the generation and usage of environment models to navigate around the racing track, roughly resembles the approaches of Andre and Brave (Section 2.2). This revolves around the notion of a *multi-phasic* fitness evaluation procedure that enforces an environment model to be stored in memory during the first phase of the evaluation procedure, and the fitness of the individual to be evaluated in the second phase, biasing individuals towards an effective storage and retrieval of information via a sensory deprivation method. We will refer to these two phases of fitness evaluation as *model-building* phase, and *car-controlling* phase respectively.

Each individual evolved program employs a modular architecture that is designed to consist of two main expression-tree branches (i.e. a model-building branch, and a car-controlling branch), one of which will be executed in each phase of the fitness evaluation procedure. This modular structure of expression-trees is coupled with a general-purpose 2-dimensional data-structure (i.e. a 2-dimensional array of `double` values), representing program's memory.

In order to build a model of the track, the controller is required to access a representation of the track itself. The track representation is based on information that has been gathered offline, while racing a hand-coded controller around the track [1]. The information is tabular, taking the form of an external input file; each file-line represents sensory information that is recorded in every simulated time-step:

1. **X**: Distance of car from the right most edge of the track. To calculate this, we combined the information of the distance between the car and the track axis (given by `getTrackPosition()`) with the information of the track-width, so that the value of zero denotes that the car is on the right-most edge of the track. This can be taken for the value of x in an x-y coordinate system, where x measures the track-width, and y measures the distance from the track origin that is taken to be the race start-line along the track-line. Figure 3 illustrates this simple coordinate system.

2. **Y**: Distance of car from the start line (given by `getDistanceFromStartLine()`). This can be taken for the value of y in an x-y coordinate system, where x measures the track-width, and y measures the distance from the track origin that is taken to be the race start-line along the track-line.

3. **Angle**: Angle between the car direction and the direction of the track-axis (given by `getAngleToTrackAxis()`).

[1] We have used `champ2011client.SimpleDriver` that is included in the distribution provided by `http://cig.ws.dei.polimi.it/`

4. **TrackEdgeSensorsA**: This is the mean value of 6 range finder sensors with indices in the interval [0, 5] that are given by `getTrackEdgeSensors()`.

5. **TrackEdgeSensorsB**: This is the mean value of 6 range finder sensors with indices in the interval [6, 11] that are given by `getTrackEdgeSensors()`.

6. **TrackEdgeSensorsC**: This is the mean value of 7 range finder sensors with indices in the interval [12, 18] that are given by `getTrackEdgeSensors()`.

Given this sensory information, each line of the input file consists of the 6-tuple (X, Y, Angle, TrackEdgeSensorsA, TrackEdgeSensorsB, TrackEdgeSensorsC). In the first phase, the individual may examine this sensory information, and may manipulate (i.e. read, write) its data-structure. The model-building branch is iteratively executed once for each 6-tuple until the whole of the input file has been exhausted. Program's memory is being initialised in the beginning of the model-building phase, and is preserved during the series of model-building branch executions with consecutive 6-tuple inputs.

Program fitness is assigned in the second phase (car-controlling phase). Racing around the track requires executing the car-controlling branch to output driving commands. In this second phase, the sensory information is deprived, with the input vector consisting solely of the 4-tuple (X, Y, Speed, LateralSpeed) that is given in every simulated time-step. Program state, represented by the instantiation of the memory array, composes an additional input to the controller. Note that during the race, X and Y come in real-time from the simulation server, they are not the result of an offline recording as in the case of the model-building phase. The multi-phasic fitness evaluation procedure depends on this sensory deprivation method to bias the evolution of programs that are able to build a model of the environment, and utilise this model for navigation purposes. Figure 1 illustrates the information flow in model-building and car-controlling phases respectively. An automatic programming system is now faced with the task of cooperatively coevolving two distinct program components of a modular stateful program representation.

3.2 Stateful Program Representation

Program representation is based on a modular architecture that consists of two main expression-tree branches that are coupled with a general-purpose 2-dimensional *indexed memory* [1] structure of `double` values. Our implementation uses a 2-dimensional array of size 2000×2000, resulting in a significantly large state space. A memory element is indexed from (0, 0) to (1999, 1999). Each of these 2000^2 elements is a `double` value in the same range. These are initialised to the value of zero prior to the first phase of the fitness evaluation procedure. The choice of memory size has been made with a particular racing track in mind (i.e. Etrack5); applying this memory-based modelling method to other tracks would require the memory size to be appropriately calibrated. Traditionally, the idealisation of indexed memory is a memory array indexed over the integers or rounded doubles whose element values are also integers or doubles in the same range [12]. The rationale behind the choice of the memory size is to provide a memory structure that is roughly isomorphic to the racing environment. *ETrack5* is a standard TORCS racing track that has been used in the current experiments. Given

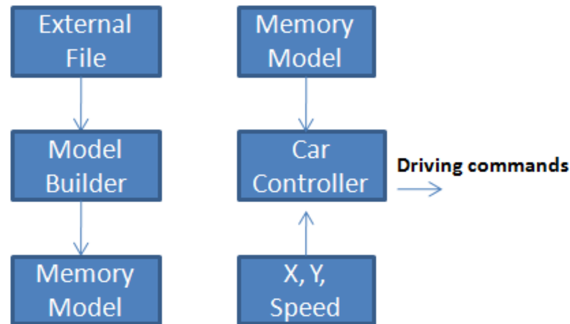

Figure 1: Information flow in model-building (left), and car-controlling (right) phases of fitness evaluation. In model-building the external file, which has been generated offline by a hand-coded driver, is used to build a model of the racing track in memory. In the car-controlling phase, the track model is used as input, along with x, y coordionates information to perform localisation.

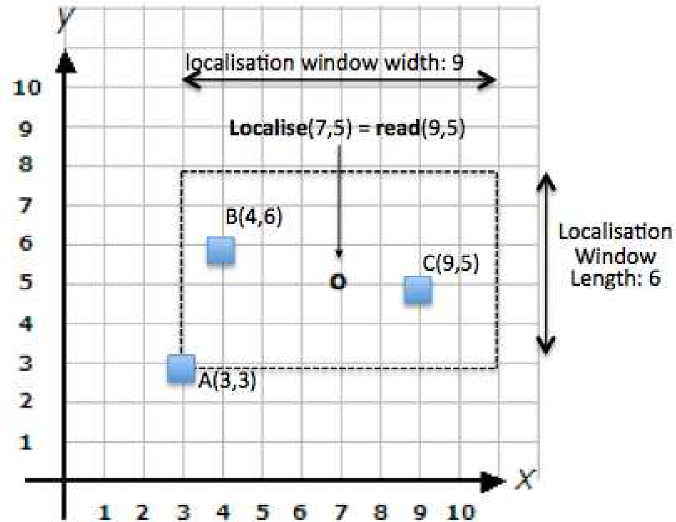

Figure 2: An example of localisation operator in 2D memory.

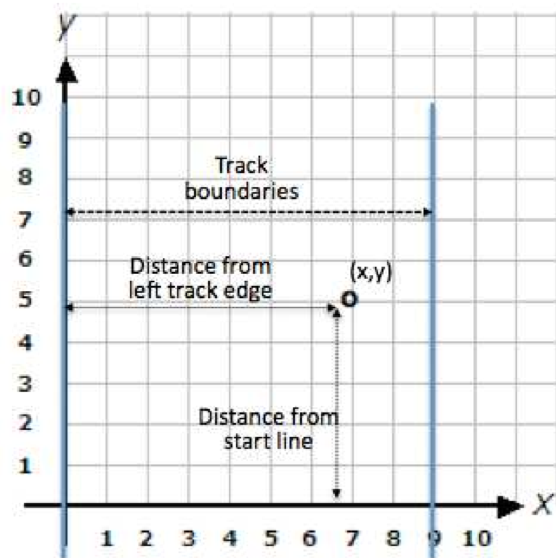

Figure 3: Illustration of the coordinate system. X refers to the distance from the left-had edge of the track. Y refers to the distance from the start line.

this environment, X and Y coordinates (discussed in the previous section) are defined within the ranges of {0,...,163} (i.e. x-axis), and {0,...,1621} (i.e. y-axis) respectively. Recall that under our normalisation scheme, 0 and 163 represent the right-most and left-most track positions respectively; whereas 0 and 1621 represent the start-line and end-of-track respectively. This design decision along with the choice of memory size have the potential of facilitating the emergence of programs that develop an isomorphic relation between the environment and the memory, by mapping an agent's move in the racing track to a roughly equivalent position in the 2-dimensional array.

Two non-terminals are added to the function-set: `read` and `write`. `read` takes two arguments x, y, and returns Memory[x][y]. `write` takes three arguments x, y, z. It returns the old value in Memory[x][y], and puts z into Memory[x][y]. Arithmetic operators are also made available in the function set (we used protected division). These functions were constrained to return double values between 0 and 1999 so that any computed value was a legal memory index. This restriction could have been relaxed by taking the index modulo 2000 before accessing memory, but was not for simplicity. Note that because the arithmetic operators and the values of memory are of type `double`, each time a memory operation is invoked, its arguments representing the array indices are rounded to the closest integer value before memory is accessed. Additionally, we introduced a fuzzy memory accessing operator by the name of `localise`. `localise` takes two arguments x, y, and returns Memory[x][y] if this memory cell has been written before; otherwise it returns the closest written memory cell. Closeness is defined in terms of Euclidean distance, where x and y access-indices are treated as x and y coordinates in the Cartesian space. Figure 2 illustrates an exam-

Table 1: Program Construction Language

Non-Terminal set	Terminal set
Model-Building Branch	
add(x, y) sub(x, y) mul(x, y) div(x, y) read(x, y) write(x, y, z)	X, Y, Angle, TrackEdgeSensorsA, TrackEdgeSensorsB, TrackEdgeSensorsC
Car-Controlling Branch	
add(x, y) sub(x, y) mul(x, y) div(x, y) read(x, y) localise(x, y)	X, Y, Speed, LateralSpeed 10 random constants in [-1.0, 1.0] interval

ple of the localisation operation. In this example, invoking `localise(7,5)` results in `read(9,5)`. This is because Memory[7][5] is empty, and Memory[9][5] is the closest non-empty memory-cell. In a different case, invoking `localise(4,6)`, will return Memory[4][6], as the requested memory cell is non-empty. For time-efficiency, we constrained the search for the closest non-empty memory cell to memory cells that are at most 150 columns and 5 rows away from the memory cell given by x, y indices. Note that in the example of Figure 2, the localisation window is of dimensions 9×6.

Table 1 summarises the primitive non-terminal and terminal language constructs that are used in model building and car-controlling branches respectively. For the case of model-building expression-tree, we are only interested in its side-effects on program's memory. The car-controlling expression-tree is designed to return two values that are instantiated by evaluating the two subtrees of its root node. The output of first subtree is interpreted as a driving (gas/break) command, whereas the output of the second subtree is interpreted as the steering (left/right) command. Both control outputs are being exponentially scaled to map to the range of the actuator values used by TORCS. The gear is set automatically depending on the rpm of the motor. For that, we have used the implementation of `getGear(SensorModel sensors)` that is available in `champ2011client.SimpleDriver` of the Java distribution, but we constrained it so that the maximum gear that can be attained is gear 1. Given this constraint, the racing car of type `car1-trb1` can develop a maximum speed of approximately 82Km/h, and can cover a distance of approximately $2,320$m within $5,000$ time-steps (distance was calculated on average based on 10 trials of a human player).

3.3 Evolutionary Algorithm

The GP system employs a generational, elitist genetic algorithm. The algorithm uses tournament selection with a tournament size of 3. The population size is set to 300 individuals, and evolution proceeds for 40 generations. Ramped-half-and-half [2] tree-

creation with a maximum depth of 5 is used to perform an initial random sampling of programs. Throughout evolution, expression-trees are allowed to grow up to depth of 8. The evolutionary search employs a variation scheme that combines mutation with standard subtree crossover. A probability governs the application of each, set to 0.7 in favour of mutation. We are combining subtree and point mutation; subtree mutation is applied with a probability 0.6 relative to point mutation. No reproduction was used. It should be noted that the crossover operator is performed only between model-building or car-controlling expression-trees, respecting the different representation language specifications (branch-typing in Koza's terminology [2]). Additionally, because of the fact that each individual is composed of two expression-trees, we apply a variation operator twice in the selected individual(s), once in each expression-tree; each application creating one offspring. This results in 600 fitness evaluations per generation. At the end of the evaluation process, the individuals are sorted in terms of fitness, and the best 300 of these form the current population.

We defined a maximisation problem, which uses a bi-objective fitness function that is reduced to a scalar value by calculating the weighted average of two objectives. The first objective is the distance raced within the allotted time-steps (the bigger the better), whereas the second objective is the average displacement from the track center during the race (the smaller the better). The second objective of the fitness function penalises individuals that use the track walls as a steering aid, and renders the emergence of sound driving behaviour more likely. Each race lasts for 5,000 simulated time-steps. The fitness function takes the following form:

$$f = w_1 * DR - w_2 * \frac{1}{5000} \sum_{i=1}^{5000} TCD_i \tag{1}$$

where DR is the distance raced (given by `getDistanceRaced()`), and TCD is the displacement from the track center in every time-step (given by `getTrackPosition()`). Both of DR and TCD are normalised within the [0.0, 1.0] interval prior calculating the fitness value. Coefficients w_1 and w_2 are used to enforce weights on the objectives. This allows for a fine-grained control over the bias that each fitness compartment should exert in the evolutionary search. We experimented with all possible combinations of these coefficients with a minimum a value of 0.5 and step size of 0.05. Herein, we report the experimental results accrued from two of the most interestingly-performing combinations of (w_1, w_2), namely (0.65, 0.35), and (0.75, 0.25).

4 Experimental Results

We present results from 25 independent evolutionary runs using each of two fitness function coefficient setups; these are (w_1, w_2) = (0.65, 0.35) and (w_1, w_2) = (0.75, 0.25). Figure 4 presents the learning curves (fitness histogram), as well as the histogram of distance raced by the best-of-generation individuals. It also illustrates the decomposition of the best-of-generation fitness value into its two compartments, and how these evolve during the course of the run. Contrasting between Figures 4(a) and 4(e), we observe that the fitness function of (0.75, 0.25) enables the GP system to learn faster

and attain higher fitness. Figures 4(b) and 4(f) further suggest that controllers are able to race longer distances under the (0.75, 0.25) coefficient setup. This result was expected, given that when a fitness function is crafted to reward long-distance racers, such behaviour should emerge. However, caution needs to be taken when interpreting the performance of the system based solely on numerical figures, and not on the actual testing and inspection of the driving behaviour in a simulation run. Inspecting how the evolved drivers behave in the simulation revealed many interesting lessons to be learnt about the nature of the search space, the effectiveness of a given fitness function coefficient configuration, and the prerequisites for efficient environment model-building.

We found that most runs that rewarded the distance-raced objective with a coefficient greater than or equal to 0.7 resulted in controllers that used the track boundaries as a driving aid to navigate around a track. A viable driving strategy that repeatedly emerged under various genotypic representations made the car to maintain a moderate constant speed while it was sliding along the track edges that were used as a steering medium. Such strategies appeared very early in the evolutionary runs, and were preserved throughout evolution, resulting in individuals that were able to cover distances up to 2,117m. This is the main reason that the learning curves of Figure 4(e) appear steeper than those of Figure 4(a) . This finding is further corroborated by a faster increase in distance raced in the case of Figure 4(f) as opposed to Figure 4(b). These results suggest that the evolutionary search could easily be misled by a fitness function that highly rewards the distance raced; most importantly such high fitness is potentially attainable without any use of an environment model for navigation.

Results indicated that the search bias, which is exerted by the second compartment of the fitness function is pivotal in order to learn how to build an environment model. Figures 4(d) and 4(h) present the evolution of the track-center-displacement objective as this is extracted from the fitness value of best-of-generation individuals, for cases (0.65, 0.35) and (0.75, 0.25) respectively. In Figure 4(d) we observe that there is an ongoing minimisation taking place, as opposed to the apparent stagnation in minimising this objective in Figure 4(h). Worse still, Figure 4(h) suggests that in the search for long-distance racers, controllers are trading a driving behaviour that exhibits on average a small displacement from the track center with a driving behaviour that results in a long distance being covered (Figure 4(g)) but does not allow for the development of an environment model. The marked tendency of the evolutionary runs with a coefficient setup of (0.65, 0.35) to optimise the second objective is attributed to the fact that a search bias has been created towards the evolution of programs that build environment models, and utilise them for navigation purposes.

We used heatmap plots as a way to visualise the program state (i.e. contents of 2-dimensional array of size 2000×2000) at the end of the model-building phase of fitness evaluation. These can also be regarded as a means of visualising read and write accesses of the array. The bottom-left corner of each heatmap corresponds to index-pair (0, 0). In the case of write-heatmaps, grey colour indicates that no value is written in the array-cell by the model-building expression-tree, while white colour indicates that the cell contains a value. For read-heatmaps, grey colour indicates that the car-controlling expression-tree is not reading from the current cell, whereas white colour indicates that the cell is being read. Memory read-accesses are allowed either by the constructs `read` or `localise`. In the case of a call to the `localise` function,

the white colour indicates the center of the search for the closest non-empty array-cell that resides within the range of 150 columns and 5 rows. Figures 5(a) and 5(b) illustrate the program state (representing the environment model after the execution of the model-building expression-tree), and the way this state is being utilised via `localise` invocations by the car-controlling expression-tree, for an evolved individual that covered an average distance of 1,031m. in 10 independent trials. To aid visualisation, we are only presenting a portion of the original array with a size of 1000×300. Inspection of the model-building expression-tree showed that the array contents in Figure 5(a) contain the evaluation of arbitrary arithmetic expressions of all three families of track-edge-sensors, and these values are extracted when `localise` is being invoked. The fact that the array is roughly isomorphic to the racing-environment allowed the evolved program to map the Y dimension of the tracing-track to individual columns in the array. Similarly, Figures 5(c) and 5(d) illustrate write and read heatmaps for an evolved individual that covered an average distance of 1,391m. during 10 independent trials. Its simplified car-controlling expression-tree is depicted in Figure 6. The subtree on the left (under the control root-node) issues driving commands, whereas the subtree on the right issues steering commands. The input arguments in the car-controller are mapped from (args[0], args[1], args[2], args[3]) to (X, Y, Speed, LateralSpeed). The steering subtree is invoking `localise(localise(0, Y), 0)`. This expression of array access maps the Y dimension of the racing track to a single column in the 2-dimensional array. Finally, Figures 5(e), 5(f), 5(g), 5(h), 5(i) illustrate program states that have been produced by various best-of-run individuals, but unfortunately haven't been successfully utilised by their respective car-controlling expression-trees. The main observation is that the model-building program components exploit the potential isomorphism between the racing-track and the memory store to map x-y environment coordinates to (x, y) array indices. One of the most neat and interesting evolved program state is depicted in Figure 5(f). Recall that the input arguments to the model-builder are mapped from (args[0], args[1], args[2], args[3], args[4], args[5]) to (X, Y, Angle, TrackEdgeSensorsA, TrackEdgeSensorsB, TrackEdgeSensorsC). In Figure 5(f), the model-building expression-tree contains the invocations `write{args[0], args[1], args[3]}` and `write{args[1], args[0], args[5]}` to map x-y coordinates to (x, y) indices, and store the values of track-edge-sensors A and C.

5 Conclusion

We presented an approach to environment model-building based on stateful modular GP. This is achieved through the cooperative coevolution of two main branches that are responsible for constructing a model based on the contents of program's memory, and hence utilising this program state to navigate around the environment. Experimental results suggest that a fitness function that penalises controllers, which deviate heavily from the track's center, provides the necessary search bias towards sound driving behaviours. We also found that most evolved environment models developed an isomorphic relation between the environment and the memory, by mapping the car's move in the racing track to a roughly equivalent position in the 2-dimensional array. From the

GP point of view, this study further corroborates the extension of the *indexed-memory* paradigm as a powerful extension that allows the evolution of effective gathering, storage and retrieval of arbitrarily complicated state information.

For future work, we will study methods that will allow the agent to utilise the evolved environment model to plan future actions. As a pilot study, we are currently experimenting with the use of multi-objective fitness functions that reward individual programs that access memory cells in greater than the current y-array-indices assuming a isomorphic relation between the array's number of rows and the environment's Y coordinate. Furthermore, we are planning to investigate the generalisability of model-building behaviours by studying the induced environment models when porting the evolved individuals in different, previously unseen racing tracks.

Acknowledgement

This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/SRC/FM1389.

References

[1] A. Teller, "Turing completeness in the language of genetic programming with indexed memory," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, vol. 1. Orlando, Florida, USA: IEEE Press, 27-29 Jun. 1994, pp. 136–141. [Online]. Available: http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/Turing.ps

[2] J. Koza, *Genetic Programming: on the programming of computers by means of natural selection*. Cambridge, MA: MIT Press, (1992).

[3] A. Agapitos, "The evolution of recursive and object-oriented programs," Ph.D. dissertation, University of Essex, Department of Computer Science and Electronic Engineering, Wivenhoe Park, Colchester, UK, 2009.

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

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

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

[7] J. Togelius, P. Burrow, and S. M. Lucas, "Multi-population competitive co-evolution of car racing controllers," in *2007 IEEE Congress on Evolutionary Computation*, D. Srinivasan and L. Wang, Eds., IEEE Computational Intelligence Society. Singapore: IEEE Press, 25-28 Sep. 2007, pp. 4043–4050.

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

[9] ——, "Multiobjective techniques for the use of state in genetic programming applied to simulated car racing," in *Proc. of IEEE CEC*, 2007, pp. 1562–1569.

[10] A. Agapitos, J. Togelius, S. M. Lucas, J. Schmidhuber, and A. Konstantinidis, "Generating diverse opponents with multiobjective evolution," in *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*. Perth, Australia: IEEE, Dec. 15-18 2008. [Online]. Available: http://julian.togelius.com/Agapitos2008Generating.pdf

[11] M. Ebner and T. Tiede, "Evolving driving controllers using genetic programming," in *IEEE Symposium on Computational Intelligence and Games, CIG 2009*, Sep. 2009, pp. 279–286.

[12] 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

[13] 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 Jun. 1994, pp. 250–255.

[14] 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 Jul. 1996, pp. 261–266.

[15] M. V. Butz and T. D. Lönneker, "Optimized sensory-motor couplings plus strategy extensions for the torcs car racing challenge," in *Proceedings of the 5th international conference on Computational Intelligence and Games*, ser. CIG'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 317–324.

[16] J. Quadflieg, M. Preuss, O. Kramer, and G. Rudolph, "Learning the track and planning ahead in a car racing controller," in *Proceedings of the 6th international conference on Computational Intelligence and Games*, ser. CIG'10, 2010.

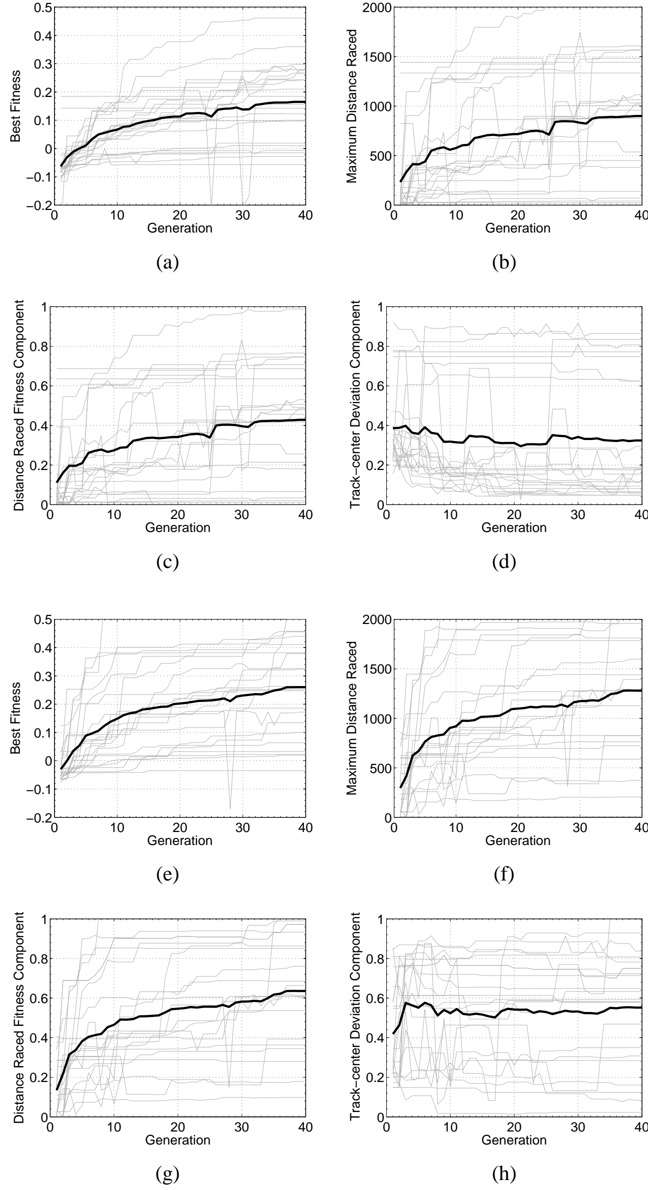

Figure 4: Best-of-generation individuals (average in bold). Column 1: Fitness histograms. Column 2: Distance-raced histograms, Column 3: Distance-raced-objective histograms. Column 4: Track-center-displacement-objective histograms. Figures (a), (b), (c), (d) for the case of (w_1, w_2) = (0.65, 0.35). Figures (e), (f), (g), (h) for the case of (w_1, w_2) = (0.75, 0.25).

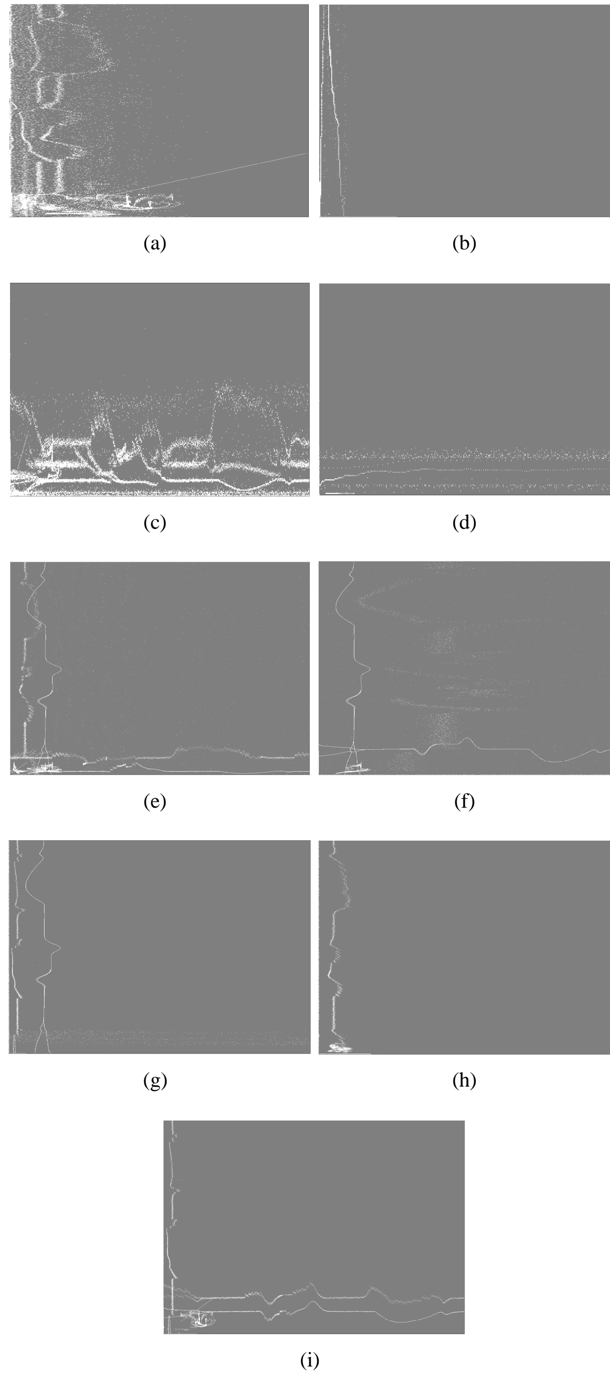

Figure 5: Heatmaps illustrating write or read accesses in the 2-dimensional `double` array. The bottom-left corner of each heatmap corresponds to index-pair (0, 0). In the case of write-heatmaps, grey colour indicates that no value is written in the cell, and white colour indicates that the cell contains a value. In the case of read-heatmaps, grey colour indicates that the car-controlling expression-tree is never reading from the current cell, and white colour indicates that the current cell is being read. In the case of function `localise`, the white cell indicates the center of the search for a non-empty cell that resides within the range of 150 columns and 5 rows. Figures (a), (b) illustrate the write and read heatmaps for an inidividual evolved under the coefficient setting (0.65, 0.35). Figures (c), (d) illustrate the write and read heatmaps for another inidividual evolved under the coefficient setting (0.65, 0.35). Figures (e), (f), (g), (h), (i) illustrate various evolved write-heatmaps. To aid visualisation, Figures (a), (b) illustrate only a part of the original array, with a size of 1000×300. Figures (c), (d) illustrate a part of the original array with a size of 200×1000. Finally, Figures (e), (f), (g), (h), (i) present a part of the array of size 1000×1000.

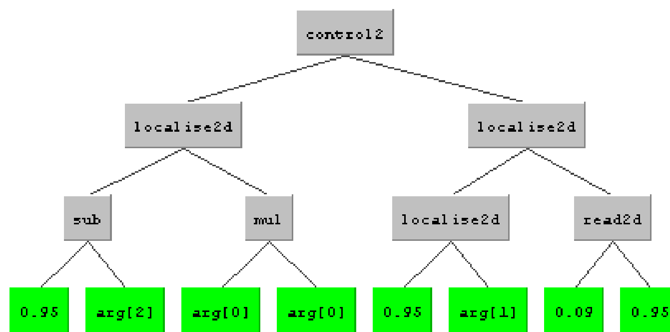

Figure 6: A simplified evolved car-controlling expression-tree that belongs to the individual program that produced read and write heatmaps depicted in Figures 5(c) and 5(d).