Computational Intelligence in Racing Games

Julian Togelius, Simon M. Lucas and Renzo De Nardi

Department of Computer Science
University of Essex
Colchester CO4 3SQ, UK
{jtogel, sml, rdenar}@essex.ac.uk

Abstract. This chapter surveys the research of us and others into applying evolutionary algorithms and other forms of computational intelligence to various aspects of racing games. We first discuss the various roles of computational intelligence in games, and then go on to describe the evolution of different types of car controllers, modelling of players' driving styles, evolution of racing tracks, comparisons of evolution with other forms of reinforcement learning, and modelling and controlling physical cars. It is suggested that computational intelligence can be used in different but complementary ways in racing games, and that there is unrealised potential for cross-fertilisation between research in evolutionary robotics and CI for games.

1 On the Roles of Computational Intelligence in Computer Games

Computational Intelligence (henceforth "CI") is the study of a diverse collection of algorithms and techniques for learning and optimisation that are somehow inspired by nature. Here we find evolutionary computation, which uses Darwinian survival of the fittest for solving problems, neural networks, where various principles from neurobiology are used for machine learning, and reinforcement learning, which borrows heavily from behaviourist psychology. Such techniques have successfully tackled many complex real-world problems in engineering, finance, the natural sciences and other fields.

Recently, CI researchers have started giving more attention to computer games. Computer games as tools and subjects for research, that is. There are two main (scientific) reasons for this: the idea that CI techniques can add value and functionality to computer games, and the idea that computer games can act as very good testbeds for CI research. Some people (like us) believe that computer games can provide ideal environments in which to evolve complex general intelligence, while other researchers focus on the opportunities competitive games provide for comparing different CI techniques in a meaningful manner.

J. Togelius et al.: *Computational Intelligence in Racing Games*, Studies in Computational Intelligence (SCI) **71**, 39–69 (2007)
www.springerlink.com

We would like to begin this paragraph by saying that conversely, many game developers have recently started giving more attention to CI research and incorporated it in their games. However, this would be a lie. At least in commercial games, CI techniques are conspicuously absent, something which CI researchers interested in seeing their favourite algorithms doing some real work often attribute to the game industry's alleged conservatism and risk aversion. Some game developers, on the other hand, claim that the output of the academic Computational Intelligence and Games ("CIG") community is not mature enough (the algorithms are too slow or unreliable), and more importantly, that the research effort is spent on the wrong problems. Game developers and academics appear to see quite different possibilities and challenges in CI for games.

With that in mind, we will provide a brief initial taxonomy of CIG research in this section. Three approaches to CI in games will be described: optimisation, innovation and imitation. The next section will narrow the focus down to racing games, and survey how each of these three approaches have been taken by us and others in the context of such games. In the rest of the chapter, we will describe our own research on CI for racing games in some detail.

1.1 Optimisation

Most CIG research takes the optimisation approach. This means that some aspect of a game is seen as an optimisation problem, and an optimisation algorithm is brought to bear on it. Anything that can be expressed as an array of parameters and where success can in some form be measured can easily be cast as presenting an optimisation problem. The parameters might be values of board positions in a board game, relative amount of resource gathering and weapons construction in a real-time strategy game, personality traits of a non-player character in an adventure game, or weight values of a neural network that controls an agent in just about any kind of game. The optimisation algorithm can be a global optimiser like an evolutionary algorithm, some form of local search, or any kind of problem-specific heuristic. Even CI techniques which are technically speaking not optimisation techniques, such as temporal difference learning (TD-learning), can be used. The main characteristic of the optimisation approach is that we know in advance what we want from the game (e.g. highest score, or being able to beat as many opponents as possible) and that we use the optimisation algorithm to achieve this.

In the literature we can find ample examples of this approach taken to different types of games. Fogel [1] evolved neural networks for board evaluation in chess, and Schraudolph [2] similarly optimised board evaluation functions, but for the game Go and using TD-learning; Lucas and Runarsson [3] compared both methods. Moving on to games that actually require a computer to play (computer games proper, rather than just computerised games) optimisation algorithms have been applied to many simple arcade-style games such as Pacman [4], X-pilot [5] and Cellz [6]. More complicated games

for which evolutionary computation have been used to optimise parameters include first-person shooters such as Counter-Strike [7], and real-time strategy games such as Warcraft [8, 9].

Beyond Optimization There is no denying that taking the optimisation approach to CI in games is very worthwhile and fruitful for many purposes. The opportunity many games present for tuning, testing and comparison of various CI techniques is unrivalled. But there are other reasons than that for doing CIG research. Specifically, one might be interested in doing synthetic cognitive science, i.e. studying the artificial emergence of adaptive behaviour and cognition, or one might be interested in coming up with applications of CIG that work well enough to be incorporated in actual commercial computer games. In these cases, we will likely need to look further than optimisation.

From a cognitive science point of view, it is generally less interesting to study the optimal form of a particular behaviour whose general form has been predetermined, than it is to study the emergence (or non-emergence) of new and different behaviours. Furthermore, something so abstract and disembodied as an optimal set of construction priorities in an RTS game doesn't seem to tell us much about cognition; some cognitive scientists would argue that trying to analyse such "intelligence" leads to the *symbol grounding problem*, as the inputs and outputs of the agent is not connected to the (real or simulated) world but only to symbols whose meaning has been decided by the human designer [10].

From the point of view of a commercial game designer, the problem with the optimisation approach is that for many types of games, there isn't any need for better-performing computer-controller agents. In most games, the game can easily enough beat any human player. At least if the game is allowed to cheat a bit, which is simple and invisible enough to do. Rather, there is a need for making the agents' behaviour (and other aspects of the game) more *interesting*.

1.2 Innovation

The border between the optimisation and innovation approaches is not clear-cut. Basically, the difference comes down to that in the optimisation approach we know what sort of behaviour, configuration or structure we want, and use CI techniques to achieve the desired result in an optimal way. Taking the innovation approach, we don't know exactly what we are looking for. We might have a way of scoring or rewarding good results over bad, but we are hoping to create lifelike, complex or just generally interesting behaviours, configurations or structures rather than optimal ones. Typically an evolutionary algorithm is used, but it is here treated more like a tool for search-based design than as an optimiser.

If what is evolved is the controller for an agent, an additional characteristic of the innovation approach is that a closed sensorimotor loop is desired:

where possible, the controller acts on sensor inputs that are "grounded" in the world, and not on pre-processed features or categorisations of the environment, and outputs motor commands rather than abstract parameters to be processed by some hard-coded behaviour mechanism [10]. To make this point about concreteness a bit more concrete, a controller for an first-person shooter that relies on pure (simulated) visual data, and outputs commands to the effect of whether to move left or right and raise or lower the gun, would count as part of a closed sensorimotor loop; a controller that takes as inputs categorizations of situations such as "indoors" and "close to enemies" and outputs commands such as "hide" and "frontal assault" would not. The point is that the perceptions and possible behaviours is influenced and limited by human preconceptions to a much larger extent in the latter example than in the former.

One of the most innovative examples of the innovation approach in a complete computer game is Stanley et al.'s NERO game. Here, real-time evolution of neural networks provide the intelligence for a small army of infantry soldiers [11] The human player trains the soldiers by providing various goals, priorities and obstacles, and the evolutionary algorithm creates neural networks that solve these tasks. Exactly how the soldiers solve the tasks is up to the evolutionary algorithm, and the game is built up around this interplay between human and machine creativity.

1.3 Imitation

In the imitation approach, supervised learning is used to imitate behaviour. What is learned could be either the player's behaviour, or the behaviour of another game agent. While supervised learning is a huge and very active research topic in machine learning with many efficient algorithms developed, this does not seem to have spawned very much research in the imitation approach to CIG. But there are some notable examples among commercial games, such as the critically acclaimed Black and White by Lionhead Studios. In this game, the player takes the role of a god trying to influence the inhabitants of his world with the help of a powerful creature. The creature can be punished and rewarded for its actions, but will also imitate the player's action, so that if the player casts certain spells in certain circumstances, the creature will try to do the same to see whether the player approves of it. Another prominent example of imitation in a commercial game is Forza Motorsport, to be discussed below.

1.4 Are Computer Games the Way Forward for Evolutionary Robotics?

When taken towards mobile robotics rather than games problems, the innovation approach is called evolutionary robotics (ER). This research field, which is 15 years or so old, aims to use evolutionary algorithms to develop as complex

and functional robot controllers as possible while incorporating as little human domain knowledge and as few assumptions as possible [12]. The problem with evolutionary robotics is that the most complex and functional robot controllers developed so far are just not very impressive. Common examples of tasks solved successfully are on the order of complexity of determining whether and how to approach one light source or the other based on the feedback gathered last time a light source was approached; similar to the type of problems Skinner subjected his rats to in his (in)famous Skinner boxes. Even very recent examples in collective evolutionary robotics [13, 14], are limited to the evolution of simple consensus strategies or minimal signalling. The evolved controllers, most often neural networks, are on the order of complexity of ten or twenty neurons, which makes for a rather diminutive cognitive faculty compared to those of garden snails and house flies. Not to mention those of yourself or the authors.

There are probably many reasons for this failure of ER to scale up, and we will be content with mention those we think are the most important, without going into detail. First of all, robotics hardware is expensive and slow, severely limiting the applicability of evolutionary algorithms, which often need thousands of fitness evaluations to get anywhere and might well destroy the robot in the course of its exploration of fitness space. Therefore, much evolutionary robotics research go on in simulation. However, the simulators are often not good enough: the evolved robot controllers don't transfer well to the real robots, and the environments, tasks, and sensor inputs are not complex enough. Robots in ER typically have low-dimensional sensors, and it can be argued that complex sensor inputs are necessary for the development of real intelligence (for example, Parker argues that complex ecosystems only came into being with the development of vision during the Cambrian Explosion [15]). And without environments that are complex enough and tasks that exhibit a smooth learning curve, allowing the evolutionary process to "bootstrap" by rewarding the controller for those initial attempts that are less bad than others as well as for truly excellent performance, complex general intelligence won't evolve.

Some already existing computer games might very conceivably provide the right environments for the evolution of complex general intelligence, and the key to scaling up evolutionary robotics. Obviously, controlling agents in computer games is orders of magnitude faster and cheaper than controlling physical robots. Further, hundreds of man-years have gone into the design of many modern commercial games, providing high-resolution graphics (allowing complex sensor inputs), large complex environments, just scoring systems (allowing for good fitness functions) and tasks that are easy to start learning but ultimately require complex general intelligence to solve. Especially relevant in this context is the thinking of game designer Raph Koster, who claims that games are fun to play because we learn from them, and that the most entertaining games are those that teach us best [16]. If he is right, then some computer games are virtually tailor-made for CI research and especially

attempts at evolving complex general intelligence, and it would almost be irresponsible not to use them for that.

2 On the Use of Computational Intelligence in Racing Games

Racing games are computer games where the goal is to guide some sort of vehicle towards a goal in an efficient manner. In its simplest form, the challenge for a player comes from controlling the dynamics of the vehicle while planning a good path through the course. In more complicated forms of racing, additional challenge comes from e.g. avoiding collisions with obstacles, shifting gears, adapting to shifting weather conditions, surfaces and visibilities, following traffic rules and last but not least outwitting competing drivers, whether this means overtaking them, avoiding them or forcing them off the track. It is thus possible for a racing game to have a smooth learning curve, in that learning to steer a slow-moving car left or right so as not to crash into a wall is not particularly hard, but on the other side of the complexity continuum, beating Schumacher or Senna at their own game seems to require a lifetime of training. Together with the potential availability of and requirement for high-dimensional sensor data, this means that it should be possible to evolve complex and relatively general intelligence in a racing game.

Let us now look at how CI can be applied to racing games:

2.1 Optimisation

Several groups of researchers have taken this approach towards racing games. Tanev [17] developed an anticipatory control algorithm for an R/C racing simulator, and used evolutionary computation to tune the parameters of this algorithm for optimal lap time. Chaperot and Fyfe [18] evolved neural network controllers for minimal lap time in a 3D motocross game, and we previously ourselves investigated which controller architectures are best suited for such optimisation in a simple racing game [19]. Sometimes optimisation is multiobjective, as in our previous work on optimising controllers for performance on particular racing tracks versus robustness in driving on new tracks [20]. We will discuss this work further in sections 4 and 5.1. And there are other things than controllers that can be optimised in car racing, as is demonstrated by the work of Wloch and Bentley, who optimised the parameters for simulated Formula 1 cars in a physically sophisticated racing game [21], and by Stanley et al., who evolved neural networks for crash prediction in simple car game [22].

2.2 Innovation

In car racing we can see examples of the innovation approach to computational intelligence in work done by Floreano et al. [23] on evolving active vision,

work which was undertaken not to produce a controller which would follow optimal race-lines but to see what sort of vision system would emerge from the evolutionary process. We have ourselves studied the effect of different fitness measures in competitive co-evolution of two cars on the same tracks, finding that qualitatively different behaviour can emerge depending on whether controllers are rewarded for relative or absolute progress [24], work which will be described in section 5.2. Proving that the innovation approach can be applied to other aspects of racing games than controllers, we have worked on evolving racing tracks for enhanced driving experience for particular human players [25, 26], work which will be presented in section 9.

2.3 Imitation

A major example of the imitation approach to computational intelligence in racing games is the XBox game Forza Motorsport from Microsoft Game Studios. In this game, the player can train a "drivatar" to play just like himself, and then use this virtual copy of himself to get ranked on tracks he doesn't want to drive himself, or test his skill against other players' drivatars. Moving from racing games to real car driving, Pomerleau's work on teaching a real car to drive on highways through supervised learning based on human driving data is worth mentioning as an example of the imitation approach [27]. The reason for using imitation rather than optimisation in this case was probably not that interesting driving was preferred to optimal driving, but rather that evolution using real cars on real roads would be costly. Our own work on imitating the driving styles of human players was published in [25] and [26] and is discussed in section 8.

3 Our Car Racing Model

The experiments described below were performed in a 2-dimensional simulator, intended to qualitatively if not quantitatively, model a standard radio-controlled (RC) toy car (approximately $17cm$ long) in an arena with dimensions approximately 3×2 meters, where the track is delimited by solid walls. The simulation has the dimensions $400 \times 300 pixels$, and the car measures $20 \times 10 pixels$.

RC toy car racing differs from racing full-sized cars in several ways. One is the simplified, "bang-bang" controls: many toy RC cars have only three possible drive modes (forward, backward, and neutral) and three possible steering modes (left, right and center). Other differences are that many toy cars have bad grip on many surfaces, leading to easy skidding, and that damaging such cars in collisions is nigh impossible due to their low weight.

In our model, a track consists of a set of walls, a chain of waypoints, and a set of starting positions and directions. When a car is added to a track in one of the starting positions, with corresponding starting direction, both the

position and angle being subject to random alterations. The waypoints are used for fitness calculations.

The dynamics of the car are based on a reasonably detailed mechanical model, taking into account the small size of the car and bad grip on the surface, but is not based on any actual measurement [28, 29]. The collision response system was implemented and tuned so as to make collisions reasonably realistic after observations of collisions with our physical RC cars. As an effect, collisions are generally undesirable, as they may cause the car to get stuck if the wall is struck at an unfortunate angle and speed.

Still, there are differences between our model and real RC cars. Obviously, parameters such as acceleration and turning radius don't correspond exactly to any specific RC car and surface combination, as is the case with the skidding behaviour. A less obvious difference is that our model is symmetric in the left/axis, while most toy cars have some sort of bias here. Another difference has to do with sensing: as reported in Tanev et al. as well as [19], if an overhead camera is used for tracking a real car, there is a small but not unimportant lag in the communication between camera, computer and car, leading to the controller acting on outdated perceptions. Apart from that, there is often some error in estimations of the car's position and velocity from an overhead camera. (Though this is significantly less of a problem if using a more sophisticated motion capture system, as discussed in section 10.) In contrast, the simulation allows instant and accurate information to be fed to the controller.

4 Evolving Controllers for the Basic Racing Task

In our first paper on evolutionary car racing, we investigated how best to evolve controllers for a single car racing around a single track as fast as possible [19]. Our focus was on what sort of information about the car's state and environment was needed for effective control, and how this information and the controller should be represented. The fitness function and evolutionary algorithm was kept fixed for all experiments, and is described below. We also discuss which approaches turned out to work and which didn't, and what we can learn from this.

4.1 Evolutionary Algorithm, Neural Networks, and Fitness Function

The evolutionary algorithm we used in these experiments, which is also the basis for the algorithm used in all subsequent experiments in this chapter, is a 50+50 evolution strategy. It's workings are fairly simple: at the beginning of an evolutionary run, 100 controllers are initialised randomly. Each generation, each controller is tested on the task at hand and assigned a fitness value, and the population is sorted according to fitness. The 50 worst controllers are

then discarded, and replaced with copies of the 50 best controllers. All the new controllers are mutated, meaning that small random changes are made to them; exactly what these random changes consist in is dependent on the controller architecture. For the experiments in this chapter, most evolutionary runs lasted 100 generations.

Three of the five architectures in this section, including the winning architecture, are based on neural networks. The neural networks are standard multi-layer perceptrons, with n input neurons, a single layer of h hidden neurons, and two output neurons, where each neuron implements the $tanh$ transfer function. At each time step, the inputs as specified by the experimental setup is fed to the network, activations are propagated, and the outputs of the network are interpreted as actions that are used to control the car. Specifically, an activation of less than -0.3 of output 0 is interpreted as backward, more than 0.3 as forward and anything in between as no motor action; in the same fashion, activations of output 1 is interpreted as steering left, center or right.

At the start of an evolutionary run, the $m \times n \times 2$ connections are initialized to strength 0. The mutation operator then works by applying a different random number, drawn from a gaussian distribution around zero with standard deviation 0.1, to the strength of each connection.

As for the fitness calculation, a track was constructed to be qualitatively similar to a tabletop racing track we have at the University of Essex (The real and model tracks can be seen in figures 1 and 2). Eleven waypoints were defined, and fitness was awarded to a controller proportionally to how many waypoints it managed to pass in one trial. The waypoints had to be passed in order, and the car was considered to have reached a waypoint when its centre was within $30 pixels$ of the centre of the waypoint. Each trial started with the car in a pre-specified starting position and orientation, lasted for 500 time steps, and awarded a fitness of 1 for a controller that managed to drive exactly one lap of the track within this time (0 for not getting anywhere at all, 2 for two laps and so on in increments of 1/11). Small random perturbations were applied to starting positions and all sensor readings.

Fig. 1. The tabletop RC racing setup

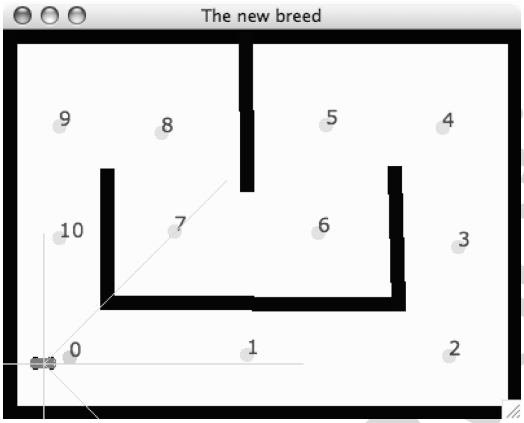


Fig. 2. The simulated racing track with the car in the starting position. The light gray lines represent wall sensors

4.2 How not to Evolve Good Driving

Four controller architectures were tried and found not to work very well, or not to work at all. These were action sequences, open-loop neural networks, neural networks with third-person information, and force fields.

Action Sequences Our action sequences were implemented as one-dimensional arrays of length 500, containing actions, represented as integers in the range 0−8. An action is a combination of driving command (forward, backward, or neutral) and steering commands (left, right or center). When evaluating an action sequence controller, the car simulation at each time step executes the action specified at the corresponding index in the action sequence. At the beginning of each evolutionary run, controllers are initialized as sequences of zeroes. The mutation operator then works by selecting a random number of positions between 0 and 100, and changing the value of so many positions in the action sequence to a new randomly selected action.

The action sequences showed some evolvability, but we never managed to evolve a controller which could complete a full lap; best observed fitness was around 0.5. Often, the most successful solutions consisted in the car doing virtually nothing for the better part of the 500 time steps, and then suddenly shooting into action towards the end. This is because each action is associated with a time step rather than a position on the track, so that a mutation early in the sequence that is in itself beneficial (e.g. accelerating the car at the start of a straight track section) will offset the actions later in the sequence in such a way that it probably lowers the fitness as a whole, and is thus selected against. As an action sequence can represent any behaviour, this controller is obviously not limited in its representational power, but apparently in its evolvability.

Open-loop Neural Networks Another attempt to evolve open-loop controllers was made with neural networks. These neural networks took only the current time step divided by the total number of time steps as input, and outputs were interpreted as above. The intuition behind this was that as multilayer perceptrons have been theoretically shown to approximate any function with arbitrary precision, this would be the case with a function from time step to optimal action. However, we didn't manage to evolve any interesting behaviour at all with this setup, indeed very few action changes were observed, and very few waypoints passed. Another case of practical nonevolvability.

Neural Networks with Third-person Information Our next move was to add six inputs to the neural network describing the state of the car at the current timestep: the x and y components of the car's position, the x and y components of its velocity, its speed and its orientation. All inputs were scaled to be in the range -10 to 10. Despite now having an almost complete state description, these controllers evolved no better than the action sequences.

Force Fields Force field controllers are common in mobile robotics research ([30]), and we wanted to see whether this approach could be brought to bear on racing cars, with all their momentum and non-holonomicity. A force field controller is here defined as a two-dimensional array of two-tuples, describing the preferred speed and preferred orientation of the car while it is in the field. Each field covers an area of $n \times npixels$, and as the fields completely tile the track without overlapping, the number of fields are $\frac{l}{n}\frac{w}{n}$, where l is length, and w is width of the track, respectively. At each time-step, the controller finds out which field the centre of the car is inside, and compares the preferred speed and orientation of that field with the cars actual speed and orientation. If the actual speed is less than the preferred, the controller issues an action containing a forward command, otherwise it issues a backward command; if the actual orientation of the car is left of the preferred orientation, the issued action contains a steer right command, otherwise it steers left. We tried various parameters but the ones that seemed least bad were fields with the size 20×20 pixels, evolved with gaussian mutation of all fields with magnitude 0.1. However, these controllers fared no better than the neural networks with third-person information. Some of the best evolved force-field controllers go half the way around the track, before the car gets stuck between two force fields, going forwards and backwards forever.

4.3 How to Evolve Good Driving

The approach that actually worked well was based on neural networks and simulated range-finder wall sensors. In this experimental setup, the five inputs to the neural network consisted of the speed of the car and the outputs of three wall sensors and one aim point sensor. The aim point sensor simply outputs

the difference between the car's orientation and the angle from the center of the car to the next aim point, yielding a negative value if that point is to the left of the car's orientation and a positive value otherwise.

Each of the three wall sensors is allowed any forward facing angle (i.e. a range of $180 degrees$), and a reach, between 0 and $100 pixels$. These parameters are co-evolved with the neural network of the controller. The sensor works by checking whether a straight line extending from the centre in the car in the angle specified by that sensor intersects with the wall at eleven points positioned evenly along the reach of the sensor, and returning a value equal to 1 divided by the position along the line which first intersects a wall. Thus, a sensor with shorter reach has higher resolution, and evolution has an incentive to optimize both reaches and angles of sensors. Gaussian random noise with standard deviation 0.01 is also added to each of the sensor readings.

Results of these experiments were excellent. Within 100 generations, good controllers with fitness of at least 2 (i.e., that managed to drive at least two laps within the allotted time) were reliably evolved, and the best controllers managed to reach fitness 3.2. We also compared the networks' performance to humans controlling the car with the keyboard, and the best evolved controller out of 10 runs narrowly beat the best human driver out of 10 human players who tried their luck at the game at the 2005 IEEE Symposium on Computational Intelligence and Games demonstration session.

A couple of variations of this setup was tested in order to explore its robustness. It was found that controllers could be evolved that reliably took the car around the track without using the waypoint sensor, solely relying on wall sensors and speed. However, these controllers drove slower, achieving fitness values around 2. (If, on the other hand, the waypoint sensor input was removed from controllers that were evolved to use it, their fitness quickly dropped to zero.) Evolving proficient controllers that used only waypoint sensor and no wall sensors was only possible when all randomness was removed from the simulation.

4.4 Take-home Lessons

Our first experiments clearly showed that the representation of the car's state and environment is crucial to whether a good controller can be evolved. Several controller architectures that should in theory be adequate for controlling the car around a single track proved to be nonevolvable in practice due to the lack of first-person sensor data. Interestingly, the controller architecture that won out is also the one which is arguably the most interesting to study from a cognitive science perspective, acting on relatively "raw" sensor data that would be trivially available to a suitably equipped real RC car. Subsequent unpublished experiments have indicated that the representation of the controller itself is not nearly as important as the format of the sensor data; we have managed to achieve proficient control using both genetic programming,

and through evolving tables of actions associated with points (cluster centres) in sensor-data space. In both these cases we used the same sensor setup as above.

Another important thing to note is that the existence of impermeable walls significantly changes the problem. Letting the car drive through walls results in a much simpler problem, as could be expected.

5 Evolving Controllers for More Complicated Tasks

The above experiments only concern one car at a time, starting from the same position on the same track. However, car racing gets really interesting (and starts demanding a broader range of cognitive skills) when several cars compete on several racing tracks. The good news is that the nature of the racing task is such that it is possible to gradually increase the complexity level from single-car single-track race to multi-car multi-track races.

For the experiments in this section we designed eight different tracks, presented in figure 3. The tracks are designed to vary in difficulty, from easy to hard. Three of the tracks are versions of three other tracks with all the waypoints in reverse order, and the directions of the starting positions reversed. Various minor changes were also made to the simulator between the above and below experiments (e.g. changing the collision model, and changing the number of time steps per trial to 700), which means that fitnesses cannot be directly compared between these experiments.

5.1 Generalisation and Specialisation

First, we focused on single-car, multi-track racing. In these experiments, each controller was equipped with six wall-sensors, and also received speed and waypoint sensor inputs. In some of the experiments, the position and ranges of the sensors were evolvable, in some they were fixed, as illustrated in fig. 4.

Evolving Track-specific Controllers The first experiments consisted in evolving controllers for the eight tracks separately, in order to rank the difficulty of the tracks. For each of the tracks, the evolutionary algorithm was run 10 times, each time starting from a population of "clean" controllers, with all connection weights set to zero and sensor parameters as explained above. Only weight mutation was allowed. The evolutionary runs were for 200 generations each.

The results are listed in table 1, which is read as follows: each row represents the results for one particular track. The first column gives the mean of the fitnesses of the best controller of each of the evolutionary runs at generation 10, and the standard deviation of the fitnesses of the same controllers. The next three columns present the results of the same calculations at

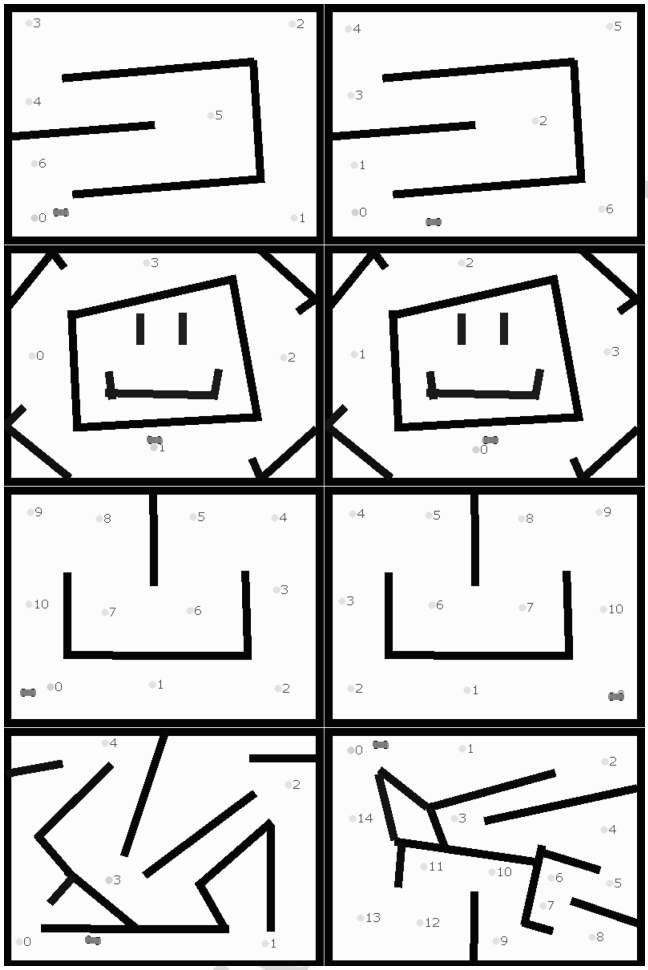


Fig. 3. The eight tracks. Notice how tracks 1 and 2 (at the top), 3 and 4, 5 and 6 differ in the clockwise/anti-clockwise layout of waypoints and associated starting points. Tracks 7 and 8 have no relation to each other apart from both being difficult

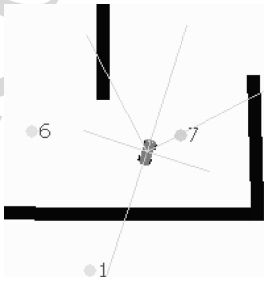


Fig. 4. The fixed sensor configuration for the incremental evolution experiments

Table 1. The fitness of the best controller of various generations on the different tracks, and number of runs producing proficient controllers. Fitness averaged over 10 separate evolutionary runs; standard deviation between parentheses

Track	10	50	100	200	*Pr.*
1	0.32 (0.07)	0.54 (0.2)	0.7 (0.38)	0.81 (0.5)	2
2	0.38 (0.24)	0.49 (0.38)	0.56 (0.36)	0.71 (0.5)	2
3	0.32 (0.09)	0.97 (0.5)	1.47 (0.63)	1.98 (0.66)	7
4	0.53 (0.17)	1.3 (0.48)	1.5 (0.54)	2.33 (0.59)	9
5	0.45 (0.08)	0.95 (0.6)	0.95 (0.58)	1.65 (0.45)	8
6	0.4 (0.08)	0.68 (0.27)	1.02 (0.74)	1.29 (0.76)	5
7	0.3 (0.07)	0.35 (0.05)	0.39 (0.09)	0.46 (0.13)	0
8	0.16 (0.02)	0.19 (0.03)	0.2 (0.01)	0.2 (0.01)	0

generations 50, 100 and 200, respectively. The "Pr" column gives the number of proficient best controllers for each track. An evolutionary run is deemed to have produced a proficient controller if its best controller at generation 200 has a fitness (averaged, as always, over three trials) of at least 1.5, meaning that it completes at least one and a half lap within the allowed time.

For the first two tracks, proficient controllers were produced by the evolutionary process within 200 generations, but only in two out of ten runs. This means that while it is possible to evolve neural networks that can be relied on to race around one of these track without getting stuck or taking excessively long time, the evolutionary process in itself is not reliable. In fact, most of the evolutionary runs are false starts. For tracks 3, 4, 5 and 6, the situation is different as at least half of all evolutionary runs produce proficient controllers. The best evolved controllers for these tracks get around the track fairly fast without colliding with walls. For tracks 7 and 8, however, we have not been able to evolve proficient controllers from scratch at all. The "best" (least bad) controllers evolved for track 7 might get halfway around the track before getting stuck on a wall, or losing orientation and starting to move back along the track.

Evolving controllers from scratch with sensor parameter mutations turned on resulted in somewhat lower average fitnesses and numbers of proficient controllers.

Generality of Evolved Controllers Next, we examined the generality of these controllers by testing their performance of the best controller for each track on each of the ten tracks. The results are presented in figure 2, and clearly show that the generality is very low. No controller performed very well on any track it had not been evolved on, with the interesting exception of the controller evolved for track 1, that actually performed better on track 3 than on the track for which it had been evolved, and on which it had a rather mediocre performance. It should be noted that both track 1 and track 3 (like all odd-numbered tracks) run counter-clockwise, and there indeed seems to be

Table 2. The fitness of each controller on each track. Each row represents the performance of the best controller of one evolutionary run with fixed sensors, evolved the track with the same number as the row. Each column represents the performance of the controllers on the track with the same number as the column. Each cell contains the mean fitness of 50 trials of the controller given by the row on the track given by the column. Cells with bold text indicate the track on which a certain controller performed best

Evo/Test	1	2	3	4	5	6	7	8
1	1.02	0.87	**1.45**	0.52	1.26	0.03	0.2	0.13
2	0.28	**1.13**	0.18	0.75	0.5	0.66	0.18	0.14
3	0.58	0.6	**2.1**	1.45	0.62	0.04	0.03	0.14
4	0.15	0.32	0.06	**1.77**	0.22	0.13	0.07	0.13
5	0.07	-0.02	0.05	0.2	**2.37**	0.1	0.03	0.13
6	1.33	0.43	0.4	0.67	1.39	**2.34**	0.13	0.14
7	**0.45**	0	0.6	0.03	0.36	0.07	0.22	0.08
8	0.16	0.28	0.09	**0.29**	0.21	0.08	0.1	0.13

AQ: Citation for Table 2 is missing. Please check.

a slight bias for the other controllers to get higher fitness on tracks running in the same direction as the track for which they were evolved.

Controllers evolved with evolvable sensor parameters turn out to generalize about as badly as the controllers evolved with fixed sensors.

Evolving Robust General Controllers The next suite of experiments were on evolving robust controllers, i.e. controllers that can drive proficiently on a large set of tracks. First, we attempted to do this by simultaneous evolution: starting from scratch (networks with all connection weights set to zero), each controller was tested on all the first six tracks, and its fitness on these six tracks was averaged. This proved not to work at all: no controllers evolved to do anything more interesting than driving straight forward. So we turned to incremental evolution.

The idea here was to evolve a controller on one track, and when it reached proficiency (mean fitness above 1.5) add another track to the training set - so that controllers are now evaluated on both tracks and fitness averaged - and continue evolving. This procedure is then repeated, with a new track added to the fitness function each time the best controller of the population has an average fitness of 1.5 or over, until we have a controller that races all of the first six tracks proficiently. The order of the tracks was 5, 6, 3, 4, 1 and finally 2, the rationale being that the balance between clockwise and counterclockwise should be as equal as possible in order to prevent lopsided controllers, and that easier tracks should be added to the mix before harder ones.

This approach turned out to work much better than simultaneous evolution. Several runs were performed, and while some of them failed to produce generally proficient controllers, some others fared better. A successful run usually takes a long time, on the order of several hundred generations, but

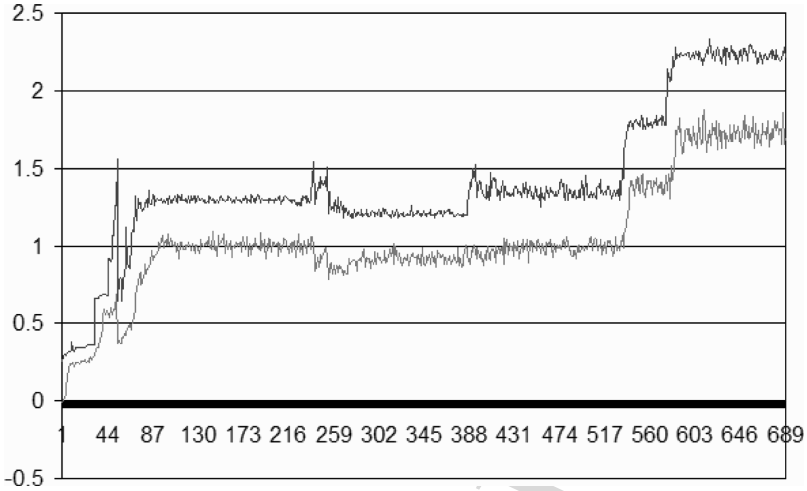


Fig. 5. A successful run of the incremental evolution of general driving skill

it seems that once a run has come up with a controller that is proficient on the first three or four tracks, it almost always proceeds to produce a generally proficient controller. One of the successful runs is depicted in figure 5 and the mean fitness of the best controller of that run when tested on all eight tracks separately is shown in 3. As can be seen from this table, the controller does a good job on the six tracks for which it was evolved, bar that it occasionally gets stuck on a wall in track 2. It never makes its way around track 7 or 8.

These runs were all made with sensor mutation turned off. Turning on sensor mutation at the start of a run of incremental evolution seems to lead to "premature specialisation", where the sensor configuration becomes specialised to some tracks and can't be used to control the car on other tracks. Evolving sensor parameters can be beneficial, however, when this is done for a controller that has already reached general proficiency. In our tests this further evolution added between 0.1 and 0.2 to the average fitness of the controller over the first six tracks.

Specialising Controllers In order to see whether we could create even better controllers, we used one of the further evolved controllers (with evolved sensor parameters) as basis for specializing controllers. For each track, 10 evolutionary runs were made, where the initial population was seeded with the general controller and evolution was allowed to continue for 200 generations. Results are shown in table 3. The mean fitness improved significantly on all six first tracks, and much of the fitness increase occurred early in the evolutionary run. Further, the variability in mean fitness of the specialized controllers from different evolutionary runs is very low, meaning that the reliability of the evolutionary process is very high. Perhaps most surprising, however, is

Table 3. Fitness of best controllers, evolving controllers specialised for each track, starting from a further evolved general controller with evolved sensor parameters

Track	10	50	100	200	*Pr.*
1	1.9 (0.1)	1.99 (0.06)	2.02 (0.01)	2.04 (0.02)	10
2	2.06 (0.1)	2.12 (0.04)	2.14 (0)	2.15 (0.01)	10
3	3.25 (0.08)	3.4 (0.1)	3.45 (0.12)	3.57 (0.1)	10
4	3.35 (0.11)	3.58 (0.11)	3.61 (0.1)	3.67 (0.1)	10
5	2.66 (0.13)	2.84 (0.02)	2.88 (0.06)	2.88 (0.06)	10
6	2.64 (0)	2.71 (0.08)	2.72 (0.08)	2.82 (0.1)	10
7	1.53 (0.29)	1.84 (0.13)	1.88 (0.12)	1.9 (0.09)	10
8	0.59 (0.15)	0.73 (0.22)	0.85 (0.21)	0.93 (0.25)	0

that all 10 evolutionary runs produced proficient controllers for track 7, on which the general controller had not been trained (and indeed had very low fitness) and for which it had previously been found to be impossible to evolve a proficient controller from scratch.

Take-home Lessons First of all, these experiments show that it is possible to evolve controllers that display robust and general driving behaviour. This is harder to achieve than good driving on a single track, but can be done using incremental evolution, where the task to be performed gradually grows more complex. This can be seen as some very first steps towards incrementally evolving complex general intelligence in computer games. Our other main result in this section, that general controllers quickly and reliably can be specialised for very good performance on particular tracks, is good news for the applicability of these techniques in computer games. For example, this method could be used to generate drivers with particular characteristics on the fly for a user-generated track.

5.2 Competitive Co-evolution

So, how about multi-car races? Our next set of experiments was concerned with how to generate controllers that drive well, or at least interestingly, in the presence of another car on the same track [24]. For this purpose, we used tracks 1, 3, and 5 from figure 3, which all run clockwise and for which it is therefore possible to evolve a proficient controller from scratch using simultaneous evolution. Each controller was evaluated on all three tracks.

Solo-evolved Controllers Together In order to characterise the difficulty of the problem, the natural first step was to place two cars, controlled by different general controllers developed using the incremental approach above, on the same track at the same time, with starting positions relatively close to each other. These two cars have only wall sensors, waypoint sensor and speed

inputs, and so have no knowledge of each other's presence. What happens when they are unleashed on the same track is that they collide with each other, and as a direct consequence of this most often either collide with a wall or lose track of which way they were going and start driving backwards. This leads to very low fitnesses. So apparently we need to evolve controllers specifically for competitive driving.

Sensors, Fitness and Evolutionary Algorithm The first step towards co-evolving competitive controllers was equipping the cars with some new sensors. In our co-evolutionary experiments each car has eight range-finder sensors, and the evolutionary process decides how many of them are wall sensors and how many are car sensors. The car sensors work just like the wall sensors, only that they report the approximate distance to the other car if in its line of detection, rather than the closest wall.

Measuring fitness in a co-evolutionary setting is not quite as straightforward as in the single-car experiments above. At least two basic types of fitness can be define: absolute and relative fitness. Absolute fitness is simply how far the car has driven along the track within the allotted time, and is the fitness measure we used in earlier experiments. Relative fitness is how far ahead of the other car the controller's car is at the end of the 700 time steps. As we shall see below, these two fitness measures can be combined in several ways.

The (single-population) co-evolutionary algorithm we used was a modified version of the evolution strategy used for earlier experiments. The crucial difference was that the fitness of each controller was determined by letting it race against three different controllers from the fitter half of the population.

Co-evolution To test the effect of the different fitness measures, 50 evolutionary runs were made, each consisting of 200 generations. They were divided into five groups, depending on the absolute/relative fitness mix used by the selection operator of the co-evolutionary algorithm: ten evolutionary runs were performed with absolute fitness proportions 0.0, 0.25, 0.5, 0.75 and 1.0 respectively. These were then tested in the following manner: the best individuals from the last generation of each run were first tested for 50 trials on all three tracks without competitors, and the results averaged for each group. Then, all five controllers in each group were tested for 50 trials each in competition against each controller of the group. See table 4 for results.

What we can see from this table is that the controllers evolved mainly for absolute fitness on average get further along the track than controllers evolved mainly for relative fitness. This applies both in competition with other cars and when alone on the track, though all controllers get further when they are alone on the track than when they are in competition with another car. Further, the co-evolved cars drive slower and get lower fitness than the solo-evolved controllers. So there are several trade-offs in effect.

What we can't see from the table, but is apparent when looking at the game running, is that the evolved controllers behave wildly differently. Most

Table 4. The results of co-evolving controllers with various proportions of absolute fitness. All numbers are the mean of testing the best controller of ten evolutionary runs for 50 trials. Standard deviations in parentheses

Proportionabsolute	0.0	0.25	0.5	0.75	1.0
Absolute fitness solo	1.99 (0.31)	2.09 (0.33)	2.11 (0.35)	2.32 (0.23)	2.23 (0.23)
Absolute fitness duo	0.99 (0.53)	0.95 (0.44)	1.56 (0.45)	1.44 (0.44)	1.59 (0.45)
Relative fitness duo	0 (0.75)	0 (0.57)	0 (0.53)	0 (0.55)	0 (0.47)

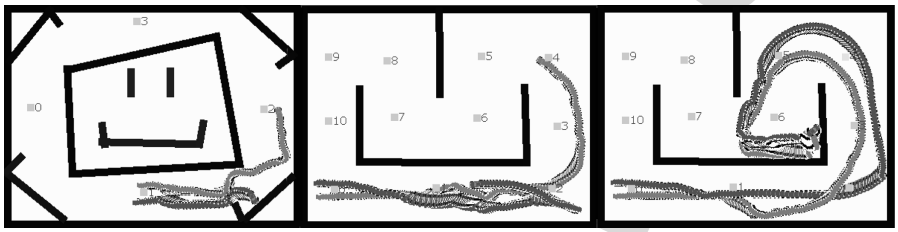


Fig. 6. Traces of the first 100 or so time-steps of three runs that included early collisions. From left to right: red car pushes blue car to collide with a wall; red car fools blue car to turn around and drive the track backwards; red and blue car collide several times along the course of half a lap, until they force each other into a corner and both get stuck. Note that some trials see both cars completing 700 time-steps driving in the right direction without getting stuck

importantly, the controllers evolved mainly for relative fitness are much more aggressive than those evolved mainly for absolute fitness. They typically aim for collisions with the other car whenever the other car stands to lose more from it (e.g. it can be forced to collide with a wall), while those evolved for absolute fitness generally avoid collision at all times. We have put some videos of such interesting car to car interactions on the web, as a video in this case says more than a thousand statistics. ([31]). See also figure 6.

We ran several auxiliary experiments using these evolved controllers, comparing different categories of controllers against each other. The most interesting result out of these comparisons is that solo-evolved controllers on average perform better than co-evolved controllers when one of each kind is placed on the same track. In fact, solo-evolved controllers perform much better against co-evolved controllers than against other solo-evolved controllers for the simple reason that the solo-evolved controllers drive faster and thus don't collide with the other car. We seem to have another trade-off here, between being able to outsmart another car and driving fast. Of course, a truly adaptive controller would recognise at what speed the other car drove and speed up if it was lagging behind, but this is not happening. At the moment we don't know whether the reason this is not happening is the simplicity of our co-evolutionary algorithm or the simplicity of the sensors and neural network we use, but we guess it's a bit of both.

6 Take-home Messages

The above example shows that the innovation approach can be taken towards the car-racing problem with good results, in that the co-evolutionary algorithm produced controllers displaying interesting and unexpected behaviour. The inability of our algorithms to produce controllers that were as general and flexible as we would have liked also demonstrates how quickly a seemingly simple problem can be turned into a highly complex problem, which might require anticipatory capabilities and complex sensing to be solved satisfactorily. This is fertile ground for future research.

7 An Application: Personalised Content Creation

In the following two sections, we will look at a possible application in games, building on the results of the above experiments. Our goal here is to automatically generate racing tracks that are fun for a particular player. The method is to model the driving style of the human player, and then evolving a new track according to a fitness function designed to measure entertainment value of the track. How to measure entertainment value is a truly complex topic which we have discussed in more depth in our publications on these experiments [25, 26]. Here we will focus on the technical issues of player modelling and track evolution.

7.1 The Cascading Elitism Algorithm

We use artificial evolution both for modelling players and constructing new tracks, and in both cases we have to deal with multiobjective fitness functions. A simple way to deal with this is to modify our evolution strategy to use multiple elites. In the case of three fitness measures, it works as follows: out of a population of 100, the best 50 genomes are selected according to fitness measure *f1*. From these 50, the 30 best according to fitness measure *f2* are selected, and finally the best 20 according to fitness measure *f3* are selected. Then these 20 individuals are copied four times each to replenish the 80 genomes that were selected against, and finally the newly copied genomes are mutated.

This algorithm, which we call Cascading Elitism, is inspired by an experiment by Jirenhed et al. [32].

8 Modelling Player Behaviour

The first step towards generating personalised game content is modelling the driving style of the player. In the context of driving games, this means learning a function from current (and possibly past) state of the car (and possibly

environment and other cars) to action, where the action would ideally be the same action as the human player would take in that situation. In other words, we are teaching a controller to imitate a human. As we shall see this can be done in several quite different ways.

8.1 What should be Modelled?

The first question to consider when modelling human driving behaviour is what, exactly, should be modelled. This is because the human brain is much more complex than anything we could currently learn, even if we had the training data available. It is likely that a controller that accurately reproduces the player's behaviour in some respects and circumstances work less well in others. Therefore we need to decide what features we want from the player model, and which features have higher priority than others.

As we want to use our model for evaluating fitness of tracks in an evolutionary algorithm, and evolutionary algorithms are known to exploit weaknesses in fitness function design, the highest priority for our model is robustness. This means that the controller does not act in ways that are grossly inconsistent with the modelled human, especially that it does not crash into walls when faced with a novel situation. The second criterion is that the model has the same average speed as the human on similar stretches of track, e.g. if the human drives fast on straight segments but slows down well before sharp turns, the controller should do the same. That the controller has a similar driving style to the human, e.g. drives in the middle of straight segments but close to the inner wall in smooth curves (if the human does so), is also important but has a lower priority.

8.2 Direct Modelling

The most straightforward way of acquiring a player model is direct modelling: log the state of the car and the action the player takes over some stretch of time, and use some form of supervised learning to associate state with action. Accordingly, this was the first approach to player modelling we tried. Both backpropagation of neural networks and nearest neighbour classification was used to map sensor input (using ten wall sensors, speed and waypoint sensor) to action, but none of these techniques produced controllers that could drive more than half a lap reliably. While the trained neural networks failed to produce much meaningful behaviour at all, the controllers based on nearest neighbour classification sometimes reproduced player behaviour very faithfully, but when faced with situations which were not present in the training data they typically failed too. Typically, failure took the form of colliding with a wall and getting stuck, as the human player had not collided with any walls during training and the sequence of movements necessary to back off from a collision was not in the training data. This points to a more general problem with direct modelling: as long as a direct model is not perfect, its

performance will always be inferior (and not just different) to the modelled human. Specifically, direct models lack robustness.

8.3 Indirect Modelling

Indirect modelling means measuring certain properties of the player's behaviour and somehow inferring a controller that displays the same properties. This approach has been taken by e.g. Yannakakis in a simplified version of the Pacman game [33]. In our case, we start from a neural network-based controller that has previously been evolved for robust but not optimal performance over a wide variety of tracks, as described in section 5.1. We then continue evolving this controller with the fitness function being how well its behaviour agrees with certain aspects of the human player's behaviour. This way we satisfy the top-priority robustness criterion, but we still need to decide on what fitness function to employ in order for the controller to satisfy the two other criteria described above, situational performance and driving style.

First of all, we design a test track, featuring a number of different types of racing challenges. The track, as pictured in figure 7, has two long straight sections where the player can drive really fast (or choose not to), a long smooth curve, and a sequence of nasty sharp turns. Along the track are 30 waypoints, and when a human player drives the track, the way he passes each waypoint is recorded. What is recorded is the speed of the car when the waypoint is passed, and the orthogonal deviation from the path between the waypoints, i.e. how far to the left or right of the waypoint the car passed. This matrix of two times 30 values constitutes the raw data for the player model.

The actual player model is constructed using the Cascading Elitism algorithm, starting from a general controller and evolving it on the test track. Three fitness functions are used, based on minimising the following differences between the real player and the controller: *f1:* total progress (number of

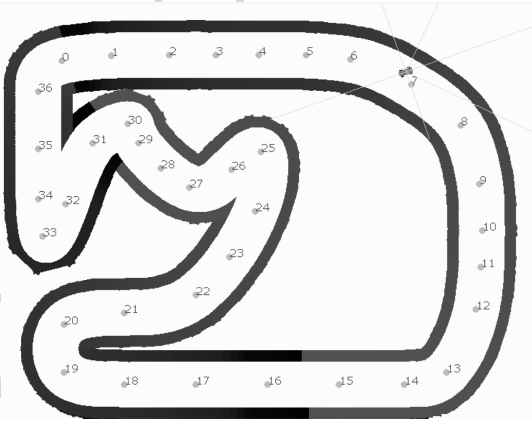


Fig. 7. The test track used for player modelling

waypoints passed within 1500 timesteps), *f2:* speed at which each waypoint was passed, and *f3:* orthogonal deviation at each waypoint as it was passed. The first and most important fitness measure is thus total progress difference, followed by speed and deviation difference respectively. Using this technique, we successfully modelled two of the authors, verifying by inspection that the acquired controllers drove similarly to the authors in question.

9 Evolving Racing Tracks

Developing a reliable quantitative measure of fun is not exactly straightforward. We have previously reasoned at some length about this, however in the experiments described here we chose a set of features which would be believed not to be too hard to measure, and designed a fitness function based on these. The features we want our track to have for the modelled player, in order of decreasing priority, is the right amount of challenge, varying amount of challenge, and the presence of sections of the track in which it is possible to drive really fast. The corresponding fitness functions are the negative difference between actual progress and target progress (in this case defined as 30 waypoints in 700 timesteps), variance in total progress over five trials of the same controller on the same track, and maximum speed.

9.1 Track Representation

The representation we present here is based on b-splines, or sequences of Bezier curves joined together. Each segment is defined by two control points, and two adjacent segment always share one control point. The remaining two control points necessary to define a Bezier curve are computed in order to ensure that the curves have the same first and second derivatives at the point they join, thereby ensuring smoothness. A track is defined by a b-spline containing 30 segments, and mutation is done by perturbing the positions of their control points.

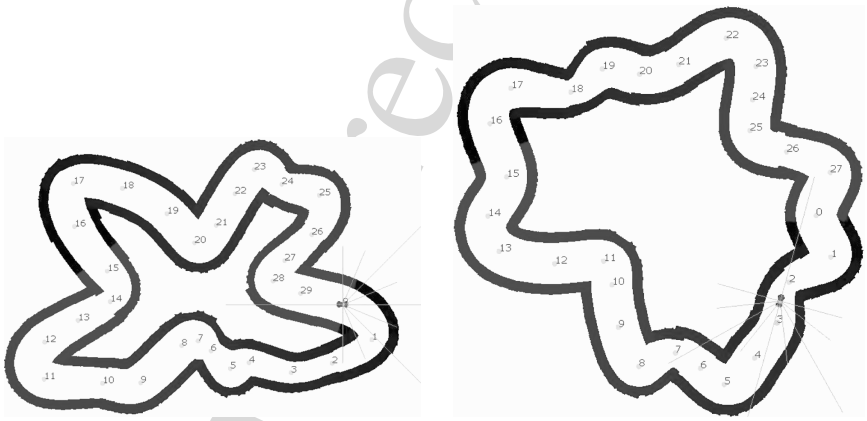
The collision detection in the car game works by sampling pixels on a canvas, and this mechanism is taken advantage of when the b-spline is transformed into a track. First thick walls are drawn at some distance on each side of the b-spline, this distance being either set to $30 pixels$ or subject to evolution depending on how the experiment is set up. But when a turn is too sharp for the current width of the track, this will result in walls intruding on the track and sometimes blocking the way. The next step in the construction of the track is therefore "steamrolling" it, or traversing the b-spline and painting a thick stroke of white in the middle of the track. Finally, waypoints are added at approximately regular distances along the length of the b-spline. The resulting track can look very smooth, as evidenced by the test track which was constructed simply by manually setting the control points of a spline.

9.2 Evolutionary Method

Each evolutionary run starts from an equally spaced radial disposition of the control points around the center of the image; the distance of each point from the center is generated randomly. Mutation works by going through all control points, and adding or subtracting an amount of distance from the center of the track drawn from a gaussian distribution. Constraining the control points in a radial disposition is a simple method to exclude the possibility of producing a b-spline containing loops, therefore producing tracks that are always fully drivable.

In figure 8 two tracks are displayed that are evolved to be fun for two of the authors. One of the authors is a significantly better driver of this particular racing game than the other, having spent too much time in front of it, and so his track has more sharp turns and narrow passages. The track that was optimised for entertaining the other player is smoother and easier to drive. In figure 9 we have plotted the progress, maximum speed, and progress variance over the evolutionary runs that produced the tracks in figure 8.

Finally, figure 10 presents a track that was evolved for the more proficient driver of the two authors using the same representation and mutation, but with a "random drift" initialisation. Here, we first do 700 mutations without selecting for higher fitness, keeping all mutations after which it is still possible for controller to drive at least one lap on the track, and retracting other mutations. After that, evolution proceeds as above.



(a) (b)

Fig. 8. Track evolved for a proficient player (a), and for a not proficient one (b) using the radial method. Although this method is biased towards "flower-like tracks", is clear that the first track is more difficult to drive, given its very narrow section an its sharp turns. A less proficient controlled instead produced an easy track with gentle turns and no narrow sections

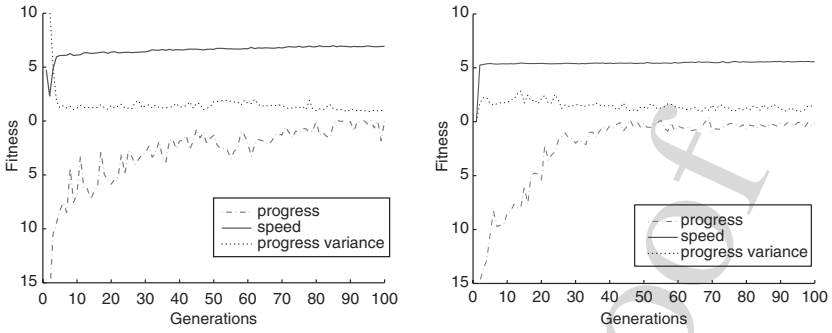


Fig. 9. Fitness graph of the evolution runs that produced the tracks in picture 8. The cascading elitism algorithm clearly optimizes for the final progress and the speed fitness. However given the interplay between progress variation and maximum speed, progress variations ends up being initially sightly reduced. This is a direct result of having speed as second (and therefore more important than the third) fitness measure in the selection process of the cascade

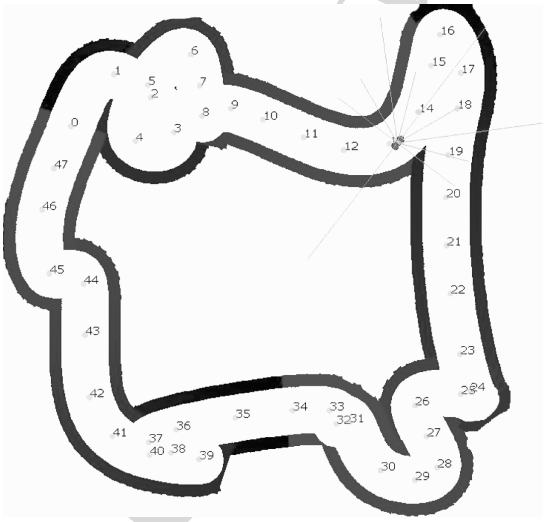


Fig. 10. Track evolved for a proficient driver using the random walk method. This track is arguably less "flower-like" than the tracks evolved without random walk initialisation

10 The Roads Ahead: Models, Reinforcements, and Physical Driving

While the experiments described above shows that good driving behaviour can be evolved, with human-competitive performance even on complex tracks, and showcased player modelling and track evolution as a potential application in

games, there are several outstanding research questions. The main question is of course exactly how complex driving behaviour it is possible to evolve. Other questions include whether we can evolve controllers for physical RC cars as well as simulated ones, and how well our evolutionary methods compare to other methods of automatically developing controllers.

Let us start with the physical car racing. We have recently set up a system where one of our inexpensive toy RC cars is controlled by a desktop PC through a standard 27 Mhz transmitter modified to work with the PC's parallel port. The car is tracked through infrared reflector markers and a not-so-inexpensive Vicon MX motion capture system, which feeds the current position and orientation of the car back to the desktop computer with very high temporal and spatial resolution and low lag (millimeters and milliseconds, respectively). Using this setup, we wrote a simple hard-coded control algorithm that could successfully do point-to-point car racing [34], which is car racing without walls and with randomised waypoints within a bounded area. Even with the excellent sensor data provided by the motion capture system, a physical car has quite a few quirks which are not straightforward to recreate in simulation. These include the response time of the motor and servos, and the variations in acceleration, turning radius and amount of drift depending on such things as what part of the floor the car is on, battery level, motor temperature and whether the car is turning left or right.

But in order to be able to evolve control for the physical car we pretty much need a model: evolving directly on the car would take prohibitively long time and might very well destroy the car in the process. One way of modelling the car would be to measure every dimension, mass, torque, friction and any other relevant detail in the car and in the track, in the attempt of deriving a model from first principles. But this would surely be painstaking, very complex and moreover it would have to be redone every time we changed cars or track. What we want is a way of automatically developing models of vehicle dynamics.

Some first steps toward this was made in a recent paper [35], where we investigated various ways of acquiring models of the *simulated* car based on first person sensor data. We tried to learn neural networks that would take the current sensor inputs at time t and the action taken at this time step, and produce the sensor vector at time $t+1$ as outputs. The results were quite interesting in that they showed that evolution and backpropagation come up with very different types of models (depending on what task you set them, evolution "cheats" in adorably creative but frequently maddening ways), but not very encouraging, in that the models we learned were probably not good enough for forming the basis of a simulator in which we could evolve controllers that could then transfer back to the system that was modelled. We think this is because we are not making any assumptions at all about the sort of function we wanted to learn, and that we could do better by explicitly introducing the basic laws of physics. The approach we are currently investigating is something we call "physics-augmented machine learning": we evolve the parameters of

a physics simulation together with a neural network, the idea being that the physics engine provides basic dynamics which are then modulated by the neural network.

But if we do manage to acquire a good model of the vehicle dynamics, could we not then use it as part of the control algorithm as well? In another recent paper [36] we compared evolution with another common reinforcement learning algorithm, temporal difference learning or td-learning. While td-learning can solve some of the same problems evolutionary computation can (but not all), it progresses in quite a different manner, in that it learns during the lifetime of the controller based on local reinforcements and not only between generations, as evolution does. In our experiments, we investigated evolution and td-learning on the point-to-point racing task with and without access to models. Generally, we found evolution to be much more reliable, robust and able to solve more variations of the problem than td-learning, but where td-learning did work it was blazingly fast and could outperform the speed of evolution by orders of magnitude. Clearly, some algorithm that combines the strengths of these two techniques would be most welcome. The comparison of learning control with and without access to a model was less ambiguous: both evolution and td-learning benefited greatly from being able to use the model to predict consequences of their actions, at least in terms of ultimate fitness of the controller. These results go well together with current thinking in embodied cognitive science, which emphasizes the role of internal simulation of action and perception in complex movements and thinking ([37]).

But we must not forget all the environments and models already there in the form of existing commercial racing games. It would be most interesting to see whether the player modelling and track evolution process would work with all the complexities and constraints of a real racing game, and using procedural techniques to generate the visual aspects of the racing track and surrounding environment would be an interesting research topic.

Finally, as controllers grow more sophisticated, we will have to look to more complex vision-like sensor representations and techniques for evolving more complex neural networks. And we will have to keep looking at findings from evolutionary robotics research to integrate into our research on computational intelligence in racing games, as well as trying to export our own findings back into that research community.

11 Resources

Currently, there is no textbook dedicated to computational intelligence in games. If you want to know more about the use of computational intelligence in games in general, a good place to start is either an edited book such as the one by Baba and Jain [38], or the proceedings of the annual IEEE Symposium on Computational Intelligence and games, which showcase the

latest work by key researchers in the field [39, 40]. For the application of computational intelligence to racing games in particular, we don't know of any authoritative overview except for this very chapter. Most of the known prior research is referenced in section 2. However, research is progressing fast with several research groups involved, and by the time you read this, new papers are probably out. We are in the process of setting up a web-based bibliography with approximately the same scope as this chapter; this should be reachable through typing the first author's name into a standard Internet search engine.

12 Acknowledgements

We thank Owen Holland, Richard Newcombe and Hugo Marques for their valuable input at various stages of the work described here.

References

1. Fogel, D.B.: Blondie24: playing at the edge of AI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
2. Schraudolph, N.N., Dayan, P., Sejnowski, T.J.: Temporal difference learning of position evaluation in the game of go. In Cowan, J.D., Tesauro, G., Alspector, J., eds.: Advances in Neural Information Processing 6. Morgan Kaufmann, San Francisco (1994) 817–824
3. Runarsson, T.P., Lucas, S.M.: Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. IEEE Transactions on Evolutionary Computation (2005) 628–640
4. Lucas, S.: Evolving a neural network location evaluator to play ms. pac-man. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games. (2005) 203–210
5. Parker, M., Parker, G.B.: The evolution of multi-layer neural networks for the control of xpilot agents. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games. (2007)
6. Togelius, J., Lucas, S.M.: Forcing neurocontrollers to exploit sensory symmetry through hard-wired modularity in the game of cellz. In: Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games CIG05. (2005) 37–43
7. Cole, N., Louis, S.J., Miles, C.: Using a genetic algorithm to tune first-person shooter bots. In: Proceedings of the IEEE Congress on Evolutionary Computation. (2004) 13945
8. Spronck, P.: Adaptive Game AI. PhD thesis, University of Maastricht (2005)
9. Miles, C., Louis, S.J.: Towards the co-evolution of influence map tree based strategy game players. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games. (2006)
10. Cliff, D.: Computational neuroethology: a provisional manifesto. In: Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats. (1991) 29–39

11. Stanley, K.O., Bryant, B.D., Miikkulainen, R.: Real-time neuroevolution in the nero video game. IEEE Transactions on Evolutionary Computation **9**(6) (2005) 653–668
12. Nolfi, S., Floreano, D.: Evolutionary robotics. MIT Press, Cambridge, MA (2000)
13. Ampatzis, C., Tuci, E., Trianni, V., Dorigo, M.: Evolution of signalling in a group of robots controlled by dynamic neural networks. In Sahin, E., Spears, W., Winfield, A., eds.: Swarm Robotics Workshop (SAB06). Lecture Notes in Computer Science (2006)
14. Baldassarre, G., Parisi, D., S., N.: Distributed coordination of simulated robots based on self-organisation. Artificial Life **12**(3) (Summer 2006) 289–311
15. Parker, A.: In the blink of an eye. Gardner books (2003)
16. Koster, R.: A theory of fun for game design. Paraglyph press (2004)
17. Tanev, I., Joachimczak, M., Hemmi, H., Shimohara, K.: Evolution of the driving styles of anticipatory agent remotely operating a scaled model of racing car. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005). (2005) 1891–1898
18. Chaperot, B., Fyfe, C.: Improving artificial intelligence in a motocross game. In: IEEE Symposium on Computational Intelligence and Games. (2006)
19. Togelius, J., Lucas, S.M.: Evolving controllers for simulated car racing. In: Proceedings of the Congress on Evolutionary Computation. (2005)
20. Togelius, J., Lucas, S.M.: Evolving robust and specialized car racing skills. In: Proceedings of the IEEE Congress on Evolutionary Computation. (2006)
21. Wloch, K., Bentley, P.J.: Optimising the performance of a formula one car using a genetic algorithm. In: Proceedings of Eighth International Conference on Parallel Problem Solving From Nature. (2004) 702–711
22. Stanley, K.O., Kohl, N., Sherony, R., Miikkulainen, R.: Neuroevolution of an automobile crash warning system. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005). (2005)
23. Floreano, D., Kato, T., Marocco, D., Sauser, E.: Coevolution of active vision and feature selection. Biological Cybernetics **90** (2004) 218–228
24. Togelius, J., Lucas, S.M.: Arms races and car races. In: Proceedings of Parallel Problem Solving from Nature, Springer (2006)
25. Togelius, J., De Nardi, R., Lucas, S.M.: Making racing fun through player modeling and track evolution. In: Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games. (2006)
26. Togelius, J., De Nardi, R., Lucas, S.M.: Towards automatic personalised content creation in racing games. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games. (2007)
27. Pomerleau, D.A.: Neural network vision for robot driving. In: The Handbook of Brain Theory and Neural Networks. (1995)
28. Bourg, D.M.: Physics for Game Developers. O'Reilly (2002)
29. Monster, M.: Car physics for games. http://home.planet.nl/ monstrous/tutcar.html (2003)
30. Arkin, R.: Behavior-based robotics. The MIT Press (1998)
31. Togelius, J.: Evolved car racing videos. http://video.google.co.uk/videoplay?docid=-3808124536098653151, http://video.google.co.uk/videoplay?docid=2721348410825414473, http://video.google.co.uk/videoplay?docid=-3033926048529222727 (2006)

32. Jirenhed, D.A., Hesslow, G., Ziemke, T.: Exploring internal simulation of perception in mobile robots. In: Proceedings of the Fourth European Workshop on Advanced Mobile Robots. (2001) 107–113
33. Yannakakis, G.N., Maragoudakis, M.: Player modeling impact on player entertainment in computer games. In: User Modeling. (2005) 74–78
34. Togelius, J., De Nardi, R.: Physical car racing video. http://video.google.co.uk/videoplay?docid=-2729700536332417204 (2006)
35. Marques, H., Togelius, J., Kogutowska, M., Holland, O.E., Lucas, S.M.: Sensorless but not senseless: Prediction in evolutionary car racing. In: Proceedings of the IEEE Symposium on Artificial Life. (2007)
36. Lucas, S.M., Togelius, J.: Point-to-point car racing: an initial study of evolution versus temporal difference learning. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games. (2007)
37. Holland, O.E.: Machine Consciousness. Imprint Academic (2003)
38. Baba, N., Jain, L.C.: Computational Intelligence in Games. Springer (2001)
39. Kendall, G., Lucas, S.M.: Proceedings of the IEEE Symposium on Computational Intelligence and Games. IEEE Press (2005)
40. Louis, S.J., Kendall, G.: Proceedings of the IEEE Symposium on Computational Intelligence and Games. IEEE Press (2006)