Evolving Race Car Drivers using Grammatical Evolution

Martin Hansen-Schwartz

School of Computer Science & Informatics
University College Dublin

Abstract

This paper will be looking at evolving a race car driver using Grammatical Evolution for the program Robot Auto Racing Simulation (RARS), a car simulation program where programmers can compete against each other by creating their own intelligent driver. The evolved driver will be based on an already existing programmed driver. The goal will be to optimize the existing driver's average speed based on 5 chosen complex tracks using Grammatical Evolution and thereby make its average speed better than it originally was. A comparison between the original driver and the evolved optimized driver shows that the evolved driver did not get to optimize the original driver due to time limitations.

1. Introduction

The aim of the introduction is to give a brief description of the program Robot Auto Racing Simulation (RARS) [1] and Grammatical Evolution (GE) [2][3].

1.1 RARS

RARS is an open source car simulation program. It uses a realistic physics model, which makes driving and steering more challenging. Each car is controlled using a drive function that is specified in each driver's .cpp file. The drive function takes situation "s" as an input and returns the car's new tyre speed and new drift angle (the angle between the car's pointing vector and its velocity vector). This is visualized in figure 1. The situation "s" has several attributes, for example, how far the car currently is from the left and right wall of the track, how fast the car is travelling and if the car is on a straight segment of the track or a right/left turn, to name a few. The car races can be displayed in 2D or 3D or the program can be run at the command line, which will return the average speed of the car. At the moment RARS works under Linux, Windows and DOS but there is no-one developing it any further as there are other car simulators that people are working on mentioned further down.

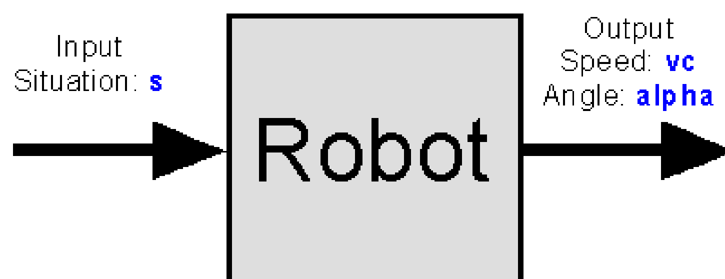

Fig. 1: Visualizing the input and output for the robot driver

1.1.1 RARS Variables

RARS has a lot of different variables that will tell different things about the situation that your car is in at any given moment during a race. You need to use these program variables to control your driver. Figure 2 gives an overview of some of the most important variables to use.

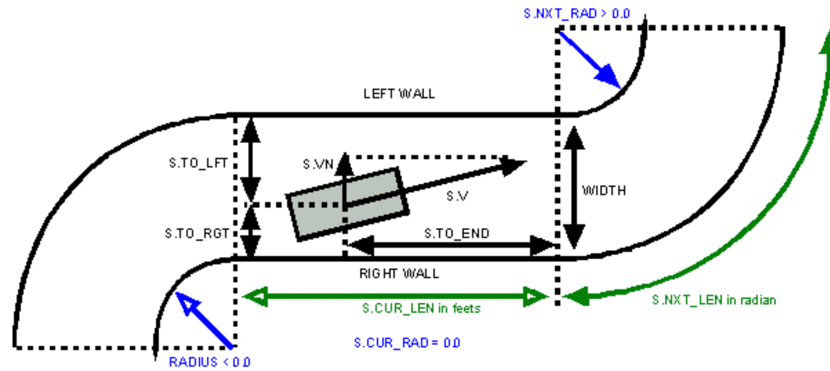

Fig. 2: Illustration of what the different RARS variables mean

Figure 2 gives you an idea of what kind of information you will need to know about when you are programming your driver. As you see from Figure 2, left turns have positive radii and right turns have negative radii. This enables you to easily distinguish between left and right turns. Straight roads have radius equal to 0. To give an example, if s.cur_rad=0 this means the car is currently on a straight road and if s.cur_rad<0 the car is in a right turn and s.cur_rad>0 the car is in a left turn. Table 1 gives a brief description of the most essential RARS variables you need to know.

Table. 1: Brief descriptions of the most essential RARS variables [4]

Variable name	Description
cur_rad	If the car is in a curve to the left, then this is the radius of the left, or inner edge of the track. If the car is in a curve to the right, then this is the negative of the radius of the right, or inner edge of the track. If on a straight road it is zero
to_end	This tells you (i.e., the robot driver) how far you have to go to reach the end of the straight or curve you are on. For a straight it is in fee and for a curve, which is a circle arc, it is in radians. You use this to decide when to begin a transition to the next segment, either by braking or accelerating, or by steering in an appropriate way.
to_lft, to_rgt	These tell you how close you are to the left and right walls of the track. If they get too small you need to steer away from the wall. In a curve, you generally want to keep near the inside wall for at least part of the curve. You do this by steering so as to maintain a small value for one of those. Finally, if either of those becomes negative, then you are off the track and accumulating damage and decelerating in speed. (These are in feet.)
v	How fast you are travelling in feet per second. Every part of the course has an appropriate speed. You need to estimate the speed required and compare your speed to it, then speed up or slow down.
vn	The component of v which is perpendicular to the track walls. This tells you if you are drifting towards a wall and if so, how quickly. You need to keep this from getting very large or very negative or you will go off the track soon afterward. Negative means heading toward right wall.
nex_rad	The radius of the next segment, zero for a straight, negative for a right turn. It is necessary to make some kind of transition as you near the end of the current segment. nex_rad lets you calculate if you need to slow down or speed up, and how much and in which direction you will have to begin heading.

1.2 Grammatical Evolution

Grammatical Evolution is a variant of Genetic Programming (GP) [5]. It builds on the same idea as GP as its objective is to find an executable program or program fragment using a given object function to find the program with the best fitness value. The components of GE consist of a grammar, an objective function and a search engine, which GE uses to produce an executable program or program fragment as mentioned previously. GE can evolve computer programs in any desired programming language. In GP (Koza-style GP [5]) any function can be a child of any other function but in GE this can be restricted as it uses a user-defined grammar (usually Backus-Naur Form), which can restrict what function calls another. The way this is done is to transcribe the binary strings (genotype) into integer strings, which

are then translated into rules defined in the grammar. The rules are then mapped into programs and are each assigned a fitness value (phenotype) using the objective function. A search-engine could be implemented with a genetic algorithm. It could also be implemented in GE with particle swarm optimization [6] used to generate and evolve a population of binary strings. These strings will then be mapped into a program that will be evaluated to find their fitness using a defined objective function. GE uses Bit Mutation, 1-point crossover and Codon Duplication as its genetic operators. Please see Figure 3 to find an overview of the mapping process of GE and an overview of the GE components.

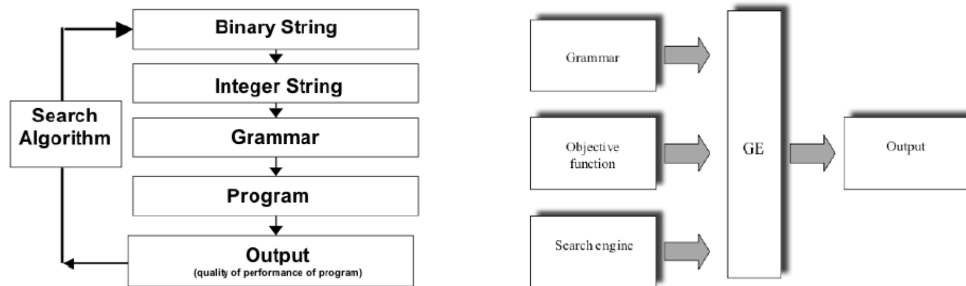

Fig. 3: An overview of the mapping process of GE (left) and the GE components (right) [7]

1.3 Other car simulators

There are also other car simulators apart from RARS. The Open Racing Car Simulator (TORCS) [8] is another popular car simulator which is directly inspired by RARS. TORCS can only be run in 3D, which is why RARS was chosen instead as RARS can be run at the command line. This means the fitness evaluation would be faster than running the graphics.

2. Experiments

The experiments were conducted using libGE [7] coupled with Evolved Objects (EO) [9]. Each experiment run was repeated 20 times. Each population consisted of 70 individuals and was evolved for 10 generations. The probability of crossover was 90% and the probability of bit-flip mutation was 1%. The random number generator used was the Mersenne Twister (eoRNG.h). The fitness for each individual was the average of the individual's average speed on five different tracks.

2.1 Test Tracks

The tracks that were used in the experiments were barcelon.trk, buenos.trk, elev2.trk, montreal.trk and spa.trk. Please look at Figure 4-8 to see how the different tracks look. The reason why these specific tracks were chosen was to try to get some complex combinations of different turns included in the fitness evaluation.

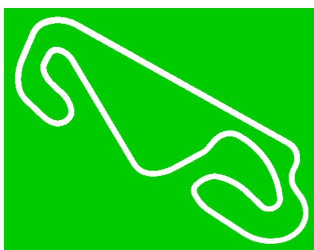

Fig. 4: barcelon.trk

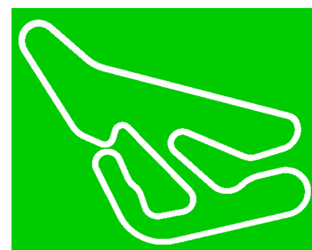

Fig. 5: buenos.trk

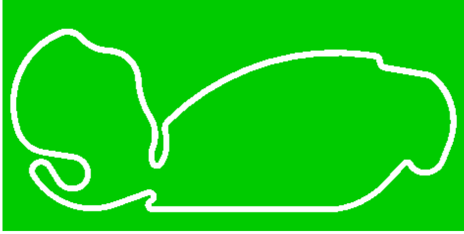

Fig. 6: elev2.trk

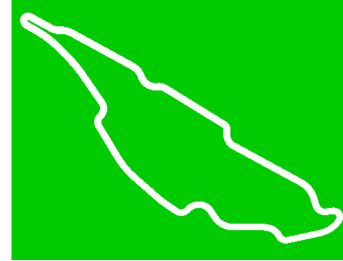

Fig. 7: montreal.trk

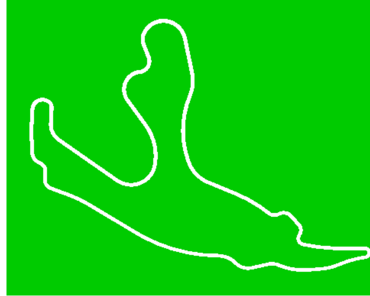

Fig. 8: spa.trk

2.2 Grammar

The grammar was based on how the robot driver Tuto3 (tutorial3.cpp, comes with the latest version of RARS) acted. Please see the grammar below in Figure 9. Some parts have been left out as it would otherwise fill up too much space. Basically there are three cases for the car. Either the car is on a straight road or else it is on a left or right turn. The grammar will decide what the alpha and vc should be. The variable "lane" will tell the car how far from the track wall it should aim to be. It has been left up to the Grammatical Evolution to find out what the constants should be. The different rules will be explained further down.

```
<start> ::= (setting up constants, left out)
if(s.cur_rad \> 0.0)\n\
    {\n\
      (determining "lane", "alpha" and "vc", left out)
    }\n\
    else if(s.cur_rad \< 0.0)\n\
    {\n\
      (determining "lane", "alpha" and "vc", left out)
    }\n\
    else if(s.cur_rad == 0.0)\n\
    {\n\
      (determining "lane", "alpha" and "vc", left out)
    }\n
<line> ::= <ifStat> | <vc> | <line> <line> |
<ifStat> ::= if(<ifCond>)\n {\n <line> }\n else\n {\n <line> }\n
           | if(<ifCond>)\n {\n <line> }\n
<ifCond> ::= <ifExpr> <logicOp> <ifExpr>
<logicOp> ::= \< | \> | ==
<alpha> ::= alpha = STEER_GAIN * (s.to_lft - lane)/width - DAMP_GAIN * s.vn/s.v +
bias;\n
<vc> ::= vc = s.v <op> <expr>;\n
<expr> ::=  <expr> <op> <val> | (<expr>) | <num>
<op> ::= + | - | * | /
<num> ::= 0.0 | 0.1 | 0.25 | 0.5 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 10.0 | 20.0
<ifExpr> ::= <ifExpr> <op> <ifVal> | (<ifExpr>) | <ifVal>
<ifVal> ::= <num> | <ifOp>
<ifOp> ::= s.cur_rad | s.v | s.nex_rad | s.to_end
         | CritDist(s.v, speed_next, BRK_CRV_ACC)
         | CritDist(s.v, speed_next, <expr>) | s.nex_len
```

Fig. 9: The grammar used for the experiments, some part have been left out

The reason why the grammar was based on a specific driver was that the search space would be too big and it would take too long to find a driver with reasonable behavior. The reason why the Tuto3 driver was chosen was because its code was easy to understand, it had a good performance and the tutorials about how to create robot drivers for RARS included the Tuto3 driver in its examples. The grammar

does not consider opponent drivers on the track, so the resulting driver will not know what to do when approaching other drivers. The decision not to consider opponent drivers was made because the command line version of RARS would only test a driver on a given track with no other drivers included. Multiple drivers could have been implemented in the RARS command line version but there was not enough time to do this. Figure 10 gives an example of an evolved driver using the grammar from Figure 9. The example has been edited slightly so it does not take up too much space in this paper.

```
static double LEFT_MARGIN = 20.0 - 3.0;
static double RIGHT_MARGIN = width - LEFT_MARGIN;
static double DELTA_MARGIN = 0.25 - 0.33 / 0.8 - 0.1;
static double START_LEFT_CORNER = RIGHT_MARGIN -(2.0);
static double START_RIGHT_CORNER = width - START_LEFT_CORNER;
static double END_LEFT_CORNER = START_RIGHT_CORNER;
static double END_RIGHT_CORNER = START_LEFT_CORNER;
static double CORNER_SLOPE = 0.7;
static double STRAIGHT_SLOPE = 0.33;
static double BIG_SLIP = 2.0;
static double BRK_CRV_ACC = -(0.25);
static double BRK_CRV_SLIP = 1.0 + 0.2;
static double BRAKE_ACCEL = -(0.1);
static double CORN_MYU = 0.25 - 0.33 / 0.8 - 0.1 / 2.0;
static double TOO_FAST = 0.7;
static double BRAKE_SLIP = 0.33;
static double CURVE_END = 2.0;
static double STEER_GAIN = 1.0 - 0.5;
static double DAMP_GAIN = 3.0 - 1.0;
if(s.cur_rad> 0.0)
{
 was_left = true;
 was_right = false;
 if(s.to_end< (temp_lane - END_LEFT_CORNER)/CORNER_SLOPE)  {
 lane = transition_lane(END_LEFT_CORNER, CORNER_SLOPE, s.to_end);}
 if(s.nex_rad< 0.0)
 { speed_next = corn_spd(fabs(s.nex_rad) + (width-START_RIGHT_CORNER), CORN_MYU); }
 else if(s.nex_rad == 0.0)
 { speed_next = 250.0; }
 speed = corn_spd(fabs(s.cur_rad) + s.to_lft, CORN_MYU);
 bias = (s.v*s.v/(speed*speed)) * atan(BIG_SLIP / speed);
 alpha = STEER_GAIN * (s.to_lft - lane)/width - DAMP_GAIN * s.vn/s.v + bias;
 to_end = s.to_end * (s.cur_rad + s.to_lft);
 if(to_end <= CritDist(s.v, speed_next, BRK_CRV_ACC))
 { vc = s.v - BRK_CRV_SLIP; }
 else if(to_end/width< CURVE_END && speed_next> speed)
 { vc = 0.5 * (s.v + speed_next)/cos(alpha); }
 else
 { vc = 0.5 * (s.v + speed)/cos(alpha); } }
else if(s.cur_rad< 0.0)
{
 was_left = false;
 was_right = true;
 if(s.to_end< (temp_lane - END_RIGHT_CORNER)/CORNER_SLOPE)
 { lane = transition_lane(END_RIGHT_CORNER, CORNER_SLOPE, s.to_end); }
 if(s.nex_rad> 0.0)
 { speed_next = corn_spd(fabs(s.nex_rad) + (width-START_LEFT_CORNER), CORN_MYU); }
 else if(s.nex_rad == 0.0)
 { speed_next = 250.0; }
 speed = corn_spd(fabs(s.cur_rad) + s.to_rgt, CORN_MYU);
 bias = -((s.v*s.v/(speed*speed)) * atan(BIG_SLIP / speed));
 alpha = STEER_GAIN * (s.to_lft - lane)/width - DAMP_GAIN * s.vn/s.v + bias;
 to_end = s.to_end * (fabs(s.cur_rad) + s.to_rgt);
 if(to_end <= CritDist(s.v, speed_next, BRK_CRV_ACC))
 { vc = s.v - BRK_CRV_SLIP; }
```

```
 else if(to_end/width< CURVE_END && speed_next> speed)
 { vc = 0.5 * (s.v + speed_next)/cos(alpha); }
 else
 { vc = 0.5 * (s.v + speed)/cos(alpha); }
 vc = s.v / 0.33;
}
else if(s.cur_rad == 0.0)
{
 if(was_left)
 { was_left = false;
 was_right = false;
 lane = LEFT_MARGIN;
 temp_lane = lane; }
 else if(was_right)
 { was_left = false;
 was_right = false;
 lane = RIGHT_MARGIN;
 temp_lane = lane; }
 if(s.nex_rad> 0.0)
 {
 if(s.to_end< (temp_lane - START_LEFT_CORNER)/STRAIGHT_SLOPE)
 { lane = transition_lane(START_LEFT_CORNER, STRAIGHT_SLOPE, s.to_end); }
 speed = corn_spd(fabs(s.nex_rad) + (width-START_LEFT_CORNER), CORN_MYU); }
 else if(s.nex_rad< 0.0)
 {
 if(s.to_end< (temp_lane - START_RIGHT_CORNER)/STRAIGHT_SLOPE)
 { lane = transition_lane(START_RIGHT_CORNER, STRAIGHT_SLOPE, s.to_end); }
 speed = corn_spd(fabs(s.nex_rad) + (width-START_RIGHT_CORNER), CORN_MYU);
 }
 else  { speed = 250.0;}
 bias = 0.0;
 alpha = STEER_GAIN * (s.to_lft - lane)/width - DAMP_GAIN * s.vn/s.v + bias;
 if(s.to_end> CritDist(s.v, speed, BRAKE_ACCEL))  {vc = s.v + 50.0;}
 else {if(s.v> TOO_FAST * speed)  {vc = s.v - BRAKE_SLIP;}
 else if(s.v< speed/TOO_FAST)  {vc = (2.0) * speed;}
 else {vc = (0.25) * (s.v + speed);}}
 if(0.2 == (0.8))  {vc = s.v / 0.33;}
}
```

Fig. 10: Example of an evolved driver using the grammar from Fig. 9

2.2.1 Genotype-Phenotype mapping example

Figure 11 gives an example of how the genotype-phenotype mapping works in GE. The example uses the <expr> rule in the grammar to demonstrate the mapping process.

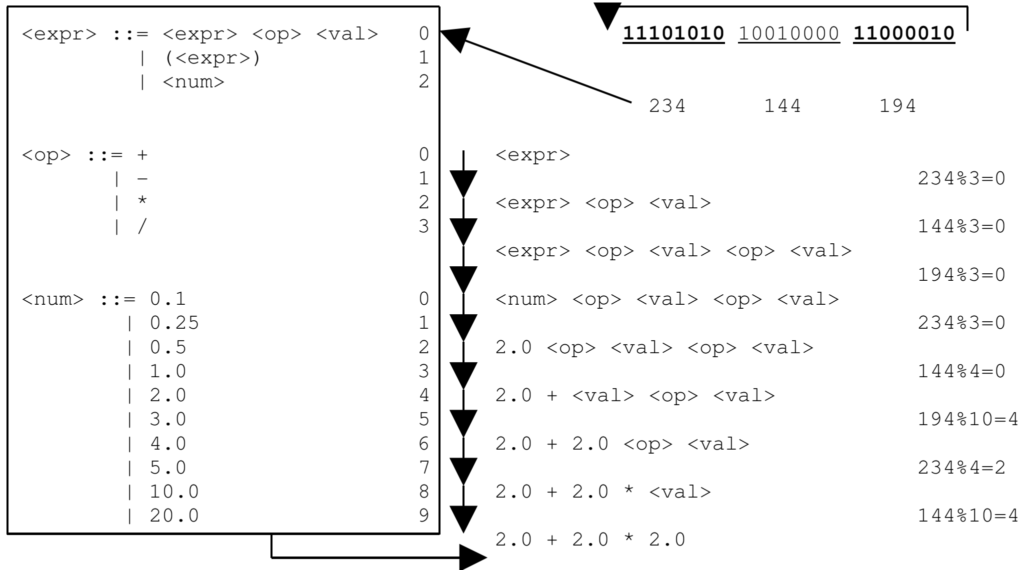

Fig. 11: The genotype-phenotype mapping process using the <expr> rule as an example. Notice the wrapping events happening when the end of binary string has been reached and mapping is not finished.

3. Results

The results did not turn out as we hoped. We will have a closer look at the results of the experiments and try to analyse them.

3.1 The average fitness at each generation

The average fitness (the average of the "average speed" of the 5 chosen tracks) at each generation is shown in Figure 12. The graph shows the overall average obtained from the average speed of 20 runs over 10 generations. The graph also shows the average speed of the Tuto3 driver. The results did not turn out as expected as the graph shows the average of average speed of the evolved driver does not even reach the average speed of the Tuto3 driver.

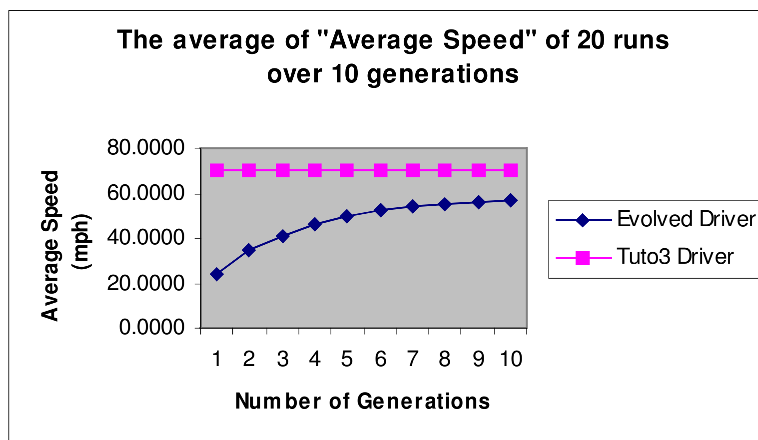

Fig. 12: The evolved driver's average fitness at each generation or " the average of the Average Speed" compared to the Tuto3 driver average speed. The average is taken from the fitness from each of the 5 tracks.

The graph in Figure 11 shows progress from generation 1 to generation 10 but the graph just seems to

flatten out before it reaches its target. It looks like the population needed more diversity to find a better solution.

3.2 Best solution found

The best solution is almost as good as that achieved by the Tuto3 driver but not better than it as was expected. Many of the other best solutions from each run were also close to the Tuto3 driver's average speed. From looking at the graph in Figure 11, the best solutions of the experiments appear to be further from the Tuto3 driver's average speed than they actually are. Figure 13 shows the best solution found at each run of the experiments which give a better idea of how close the evolved driver got to the original driver. The overall best solution was found in the 11th run and had an average speed of 69.72mph compared to the Tuto3 driver's average speed of 69.8762mph. The best solution of each run ranges between about 62 – 69mph.

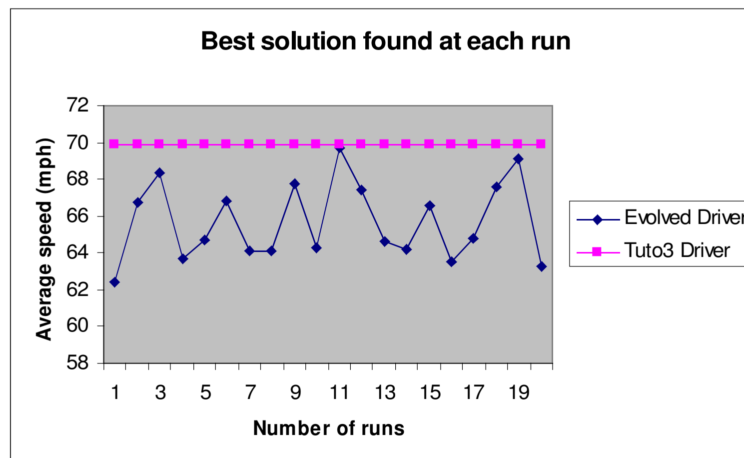

Fig. 13: The best solution found at each run of the experiments. The evolved driver almost reached the original driver's average speed but never outperformed it.

4. Future Work

As this project has been quite limited by time constraints, there are lots of other possibilities that could develop this project further.

4.1 Consider other drivers on the track

It could be interesting to see how the evolved driver would perform if you introduced opponent drivers during the fitness evaluation. This could easily be achieved by modifying the code for the command line version of RARS.

4.2 New Grammar

The grammar used for the experiments was mainly based on the Tuto3 driver. Modifying the grammar in the right way would probably give better results. Maybe adding some new features to the grammar, based on new ideas on how the car could react during the race, would give even better results. It could also be that there should be more predefined operations to make the search space smaller, but this is just a suggestion and it might not help.

4.3 Change probability of Mutation

During some of the runs the best solution of each run was already found in between 1st and the 3rd generation. This might suggest that the search needs to increase the probability of the mutation to keep the diversity in the population to avoid premature convergence.

4.3 More generations

You could try to increase the maximum number of generations and see if that would find a better solution for each generation. The experiments for this project were limited to a maximum of 10 generations as each run is very time consuming.

4.4 Larger populations

Increasing the size of the population might help find better solutions quicker. The size of the population was another limiting factor as this also increased the amount of time it would take for each run.

4.5 Specific strategies for the driver

New specific strategies for the driver could also be applied. You could try to concentrate on specific areas for example you could concentrate on the car's speed when it is turning or specific steering strategies. New strategies could also involve specific behaviour in specific situations during a race for example should the car drive more erratically at the end of a race to try to achieve a better position?

4.6 Test GE performance on individual tracks

The experiments undertaken in this project were based on the average performance of 5 different tracks. It could be interesting to see how well GE performs on the individual tracks on their own and then compare them to each other and see if there is a specific kind of track that GE performs better on.

5. Conclusions

The results of the experiments did not turn out as expected. The goal was not reached as the best solution out of all the runs was not as good as the Tuto3 driver. The reason for this might be because there was not enough time to do the experiments. This would therefore suggest that to decrease the time spent on each experiment run, the size of the population and the number generations had to be limited. The results might have turned out better if there had been a possibility to increase these two parameters. It could also be the case that the grammar should have been defined differently to give better results or maybe the search space was too big to allow for a better performance as the population size and the number of generations was limited to small numbers. As you can see in the future work section, there are a lot of opportunities to build on this project.

References

[1] Kjær, C., Teise, E., Judd, G., Pascutto, G., Guery, M., Coulom, R., Foden, T.: 1995. *RARS: Robot Auto Racing Simulation, v. 0.91_2.* SourceForge

[2] O'Neill, M., Ryan, C.: 2001. *Grammatical Evolution.* IEEE Transactions on Evolutionary Computation

[3] O'Neill, M., Ryan, C.: 2003. *Grammatical Evolution.* Kluwer Academic Publishers

[4] Timin, M.: RARS Online Tutorial 6, *http://rars.sourceforge.net/doc/tu_m6.htm*

[5] Koza, J.R.: 1992. *Genetic Programming: On the programming of computers by natural selection.* MIT Press, Cambridge, Mass

[6] Kennedy, J., Eberhart, R.: 1995. *Particle swarm optimization.* Proceedings of the IEEE International Conference on Neural Networks, Piscataway, NJ

[7] Nicolau, M.: 2004. *libGE User Manual, v. 0.26*, BDS Group, University of Limerick

[8] Wymann, B., Espie, E.: 1999. *TORCS: The Open Racing Car Simulator.* SourceForge

[9] Schoenauer, M., Merelo, J. J., Keijzer, M.: 2000. *EO: Evolving Objects, v. 0.9.3z.* SourceForge