A Genetic Algorithm in the Game Racetrack

ROBERT OLSSON
and ANDREAS TARANDI

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2011

A Genetic Algorithm in the Game Racetrack

R O B E R T   O L S S O N
a n d   A N D R E A S   T A R A N D I

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011
Supervisor at CSC was Johan Boye
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/
olsson_robert_OCH_tarandi_andreas_K11062.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

A genetic algorithm is a search heuristic that uses the principles of natural evolution to solve search and optimization problems.

In this report we describe our implementation of a genetic algorithm that can play the game racetrack. Racetrack is a game traditionally played with pen and paper. The report also goes through different optimizations and shows how they affect the performance of the algorithm.

Sammanfattning

En genetisk algoritm är en sökheurestik som använder samma principer som biologisk evolution för att lösa sök- och optimeringsproblem.

I den här rapporten beskriver vi vår implementation av en genetisk algoritm som kan spela spelet racetrack. Racetrack är ett spel som traditionellt spelas med penna och papper. Rapporten går också igenom olika optimeringar och framställer hur de påverkar algoritmens prestanda.

Contents

Statement of collaboration

We have designed the algorithm together. The game engine (except the graphics) and the algorithm part of the code are mainly written by Andreas while the graphics is mainly written by Robert. This report is written and proof read by both of us together.

1 Introduction

Artificial Intelligence, or AI, has long been used in games to create the illusion of smart opponents. In the beginning a game AI could be a predefined path for a player to use, but in today's games AI players almost seem to make their own decisions. As far as we know genetic algorithms is not something commonly used in game AI. In this report we execute experiments to investigate how well a genetic algorithm can perform in a game commonly played with pen and paper, racetrack. A detailed description of the game can be found in section 2.1.

An intelligent agent, or just an agent, is according to Russel and Norvig an AI term for "anything that can be viewed as **perceiving** its environment through **sensors and acting** upon that environment through **effectors**". [2] In our case each car in the game can be seen as an agent.

2 Problem statement

We have chosen to test the hypothesis that genetic algorithms can be used to create agents that finish tracks in the game racetrack without the agents having prior knowledge of the tracks. To test this we will design and implement an algorithm playing the game. We will also investigate how different attributes in our algorithm influences the performance.

2.1 Racetrack

Racetrack is a game classically played with pen and paper, but there are also video game implementations of it. The game is played on a track drawn on a grid. The track is the racetrack that the cars are supposed to stay inside.

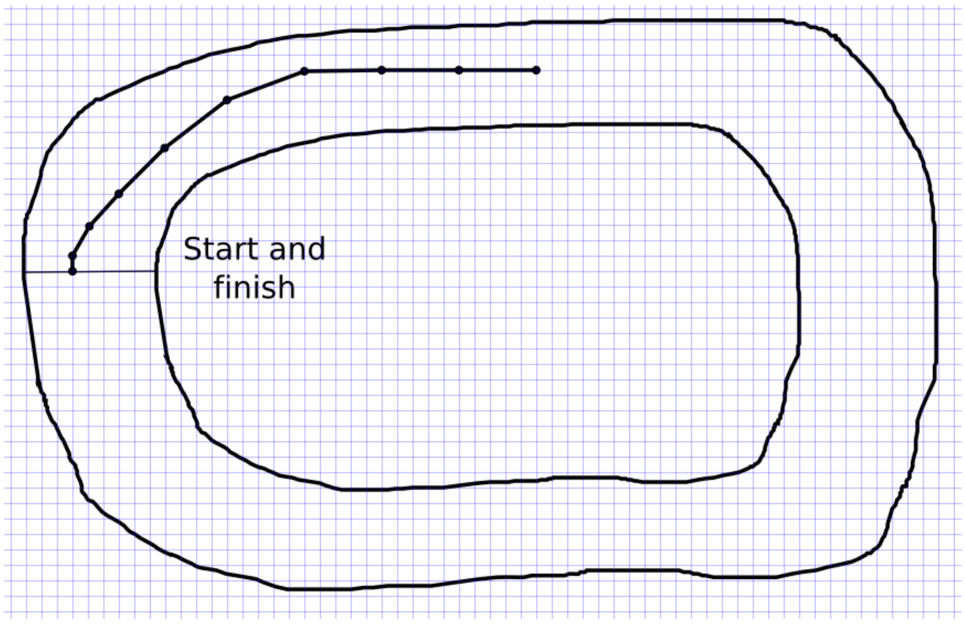

Figure 1: This is an example of a racetrack game. The path shown in the track is a player's race path.

Since the game is supposed to illustrate a real race the cars have inertia. The players take turns moving. Each movement can be seen as a vector, we call this the movement vector. The movement vector consists of two vectors; an inertia vector and an acceleration vector (see figure 2). The inertia vector is always the same as the player's previous movement vector. The acceleration vector can be any of the eight vectors in figure 3, or the null vector if the player does not apply any acceleration. [1]

If a car hits the wall of the track the player loses. The player that reaches the finish line first wins.

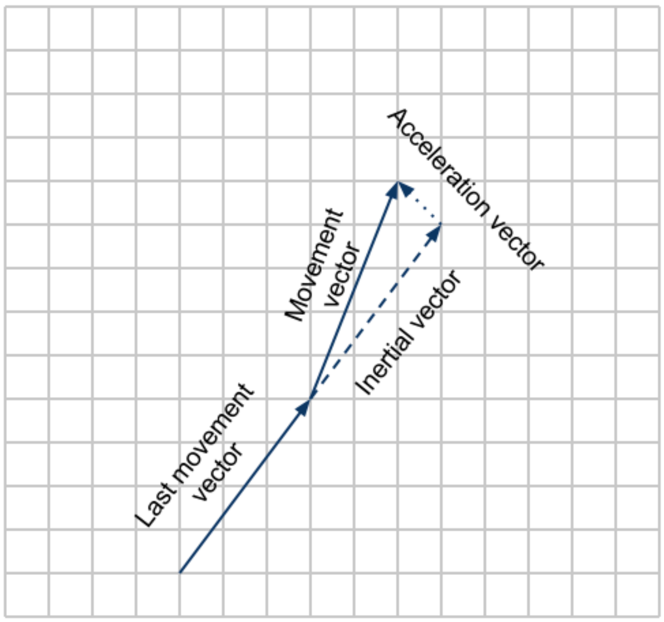

Figure 2: Here a movement vector is illustrated with its inertial and acceleration components.

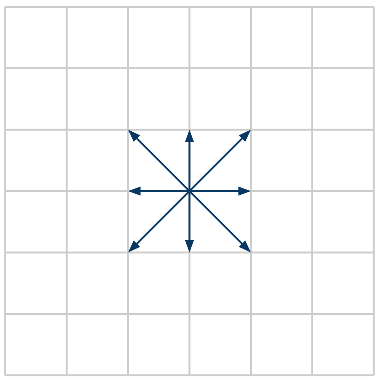

Figure 3: These are the possible acceleration vectors. The null vector is also valid.

2.2 Genetic Algorithms

To find a path from goal to start the game shall use a genetic algorithm. Our AI is supposed to learn by generating individuals that are run through a racetrack and evaluated according to some fitness function. The best performing individuals will then become the base for the next generation.

3 Background

3.1 Beam search

Local beam search is a search algorithm that runs k parallel searches, and that in each step selects the k best successors from the group of all successors. The advantages of this algorithm is that it quickly abandons unfruitful states and focus its resources where most progress is being made [2].

The disadvantages of this algorithm is clearly the lack of diversity, it may quickly become concentrated in a small region of the search space. A variant of the algorithm called **stochastic beam search** solves this by selecting k successors at random, but with the probability of a choosing a given successor being based on the value of it.

3.2 Genetic algorithm

A variant of stochastic beam search is the genetic algorithm in which successor states are generated not from one state but two. The resemblance to natural selection is clear; the strongest individuals survive and mates to produce the next generation whose genes are based on the parents'.

Like beam searches, genetic algorithms start with k randomly generated states, called the *population*. Each state or *individual* is represented by a string of a fix length over a finite alphabet, most commonly 0 and 1, the *genome*. What this string represents differs between implementation, but it must represent the full state of the individual and not change during the lifetime of an individual [2].

To produce the next generation each individual is rated by a objective function, called the *fitness function*. The best individuals are chosen and then mated, that is to say their genomes are combined to create a new individual.

This is called a crossover. Depending on the implementation the crossover is made differently, but the number of genes to use from each parent is either chosen randomly or based on the relative value of the respective fitness functions of the parents. Finally each individual is subject to random *mutation* that changes a random number of genes in their genome.

The primary advantage of genetic algorithms comes from the crossover that combines two independent states into one. This makes the algorithm converge towards a good solution quickly. The problem of all hill-climbing algorithms (algorithms that always choose the best neighboring state) is that they risk getting stuck on a local maximum that is not a solution. This is partly prevented in genetic algorithms with the use of random mutation, but they still risk getting stuck on a local maximum for a large number of generations.

4 Approach

We decided to implement our algorithm in C++. We split the implementation into two separate components, the game engine and the genetic algorithm. We did this because we wanted to be able to easily try different algorithms.

4.1 Game Engine

To be as flexible as possible the game engine is not involved in any decision making. Its task is to load track files, handle the graphics and keep track of the inertia and position of the current running individual. The game engine uses OpenGL for drawing, and has functions for drawing the track and the racing line that the agents take. All collision detection is also handled by the game engine.

4.1.1 Track format

The tracks are simply represented as images and are in our case created in Photoshop, but can be created in any graphics editing program. Different parts of the tracks are represented by different colors. The track itself is white. Red and blue denotes the start and finish zones. The starting point of the car is marked as a yellow pixel on the track image. Each track also consists of a .lvl-file with some track-specific attributes for the algorithm.

To make sure the cars go all the way around the track moving from the start to the finish zone results in a crash. Completion of the track is accomplished by going from the finish to the start zone.

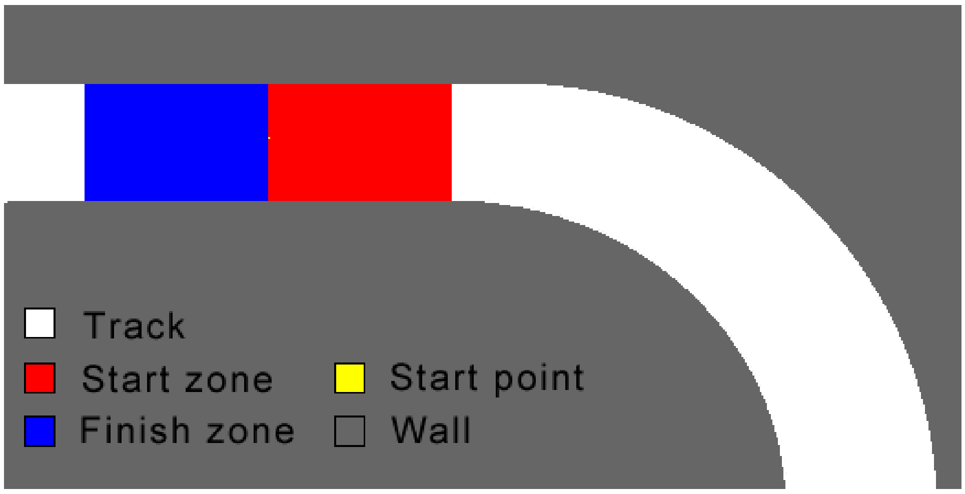

Figure 4: The different colors of the track.

4.2 Algorithm idea

For the algorithm we considered the environment for the agent to be deterministic and partly observable. The real game is actually fully observable, but to simplify the problem we have chosen to limit what the agent can see.

4.2.1 Sensors

In our algorithm the agent has three sensors; ahead, right and left (see figure 5). Each sensor returns either near, normal or far, indicating the distance to the nearest wall in the given direction. Because the thresholds of these sensor levels really are dependent of the track (how narrow the track is for example), they are specified by the *sensor_near* and the *sensor_normal* attributes in the .lvl-files.

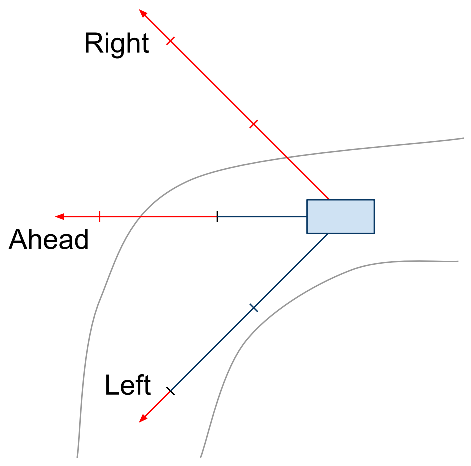

Figure 5: Illustration of the three sensors. Because of the corner in the track and the position of the car, the different sensors are reporting different distances to the wall.

4.2.2 Genome

The agent is also aware of its current inertia vector. In each state that the agent reaches it has to make a decision based on the tuple with the sensor values and the inertia. For this our algorithm looks up this tuple in a table mapping tuples to movement decisions. Because there are nine possible movements each is represented by an integer (0-8).

This table is the genome, and each individual has one. To make the size of the genome fixed we limited the speed of the agent to *max_ speed* distance units per turn in any direction, and therefore also limiting the inertia. We have used a value of 5 for *max_ speed* as it seemed like a good compromise between speed and genome size. Because of this limit, the inertia in both axes can range from -5 to 5, giving us a total of $11 * 11 = 121$ possible inertia vectors. Since we have three sensors that each can return three different values, we have a total of $3^3 = 27$ different permutations for the sensors. This gives us a total of $121 * 27 = 3267$ different state tuples. The genome must therefore contain references to 3267 movement decisions. The size of the genome is 3267 integers in the range 0-8. In our implementation we store each of these integers, or genes, in bytes making each genome 3267 bytes in size.

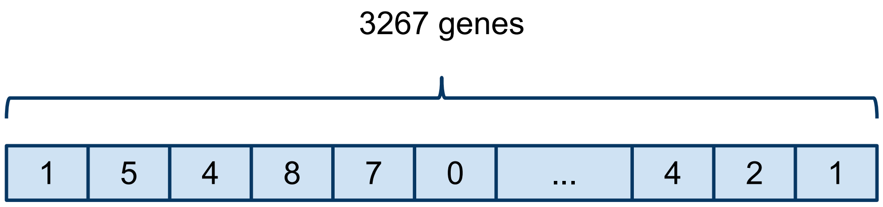

Figure 6: The genome consists of 3267 genes. Each gene is an integer in the range 0 to 8 and references a movement decision.

To find the index in the genome corresponding to the current state we use the following hash function:

$$h(inertia, sensors) = inertia.x + inertia.y * (max\_speed * 2 + 1)$$
$$+(sensors.left + sensors.ahead * 3 + sensors.right * 9) * (max\_speed * 2 + 1)^2 \qquad (1)$$

4.2.3 Crossover

When the algorithm starts it generates 100 random individuals. These individuals are then tested against the track and given a score using the fitness function. The ten individuals with the best score are then selected and each of these individuals are then crossed over with all the other selected individuals in pairs, giving 100 new individuals; the next generation.

The crossover works by sequentially going through each of the two parents' genes in pairs. At every gene we decide to pick either the first or the second parent's gene in the corresponding position in the child's gene. This decision is made randomly for each gene with a probability p_1 for the first parent and p_2 for the second. These probabilities have the same ratio as the fitness function result of the parents (but inverted since the fitness function returns lower values for better individuals), that is to say $\frac{p_1}{p_2} = \frac{fitness\_result_2}{fitness\_result_1}$. For example if the first parent got the fitness result 50 and the second parent got the result 100, we would get $p_1 = 2/3$ and $p_2 = 1/3$. The children will then have approximately 2/3 of the genes from the first parent and 1/3 of the genes from the second parent. This reduction in reproductive success for some individuals is called selection pressure. In nature the selection pressure maybe not exerted on a genome level, but this is to simulate the selection pressure that might occur during the lifetime of an individual where more fit individuals may get more reproductive chances.

During the crossover mutations can also occur. Mutations occurs on gene level for the child, and each gene has a probability of 0.05 to mutate. This is a fixed value that we have reached by some testing. When a gene mutates it just gets a random value in the range 0-8 instead of getting the value from one of its parents.

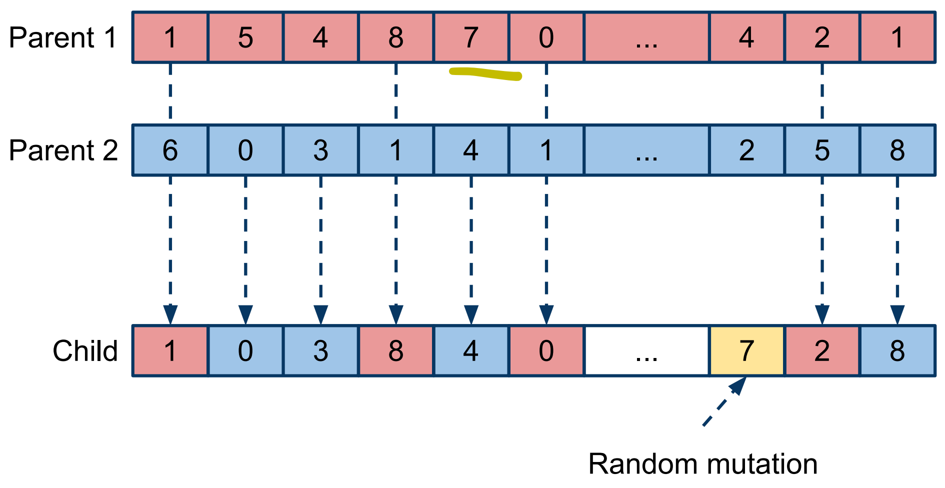

Figure 7: Example of a crossover between two individuals. A mutation is visible in the child.

The process is then repeated for the new generation; test against track, run fitness function, select 10 best individuals, crossover and mutate. This is repeated a given number of times, how many depends on how difficult the track is and is specified by the *generations* attribute in each .lvl-file.

4.2.4 The fitness function

We have chosen to judge the individuals in our problem by measuring how close to goal they came before they crashed and how many moves they required to get there. Our fitness function (f) is as follows:

?

$$f(agent\_position, num\_turns) = goalDistance(agent\_position) + num\_turns \tag{2}$$

A lower result on the fitness function means a better individual. By including the number of turns that the agent takes in the fitness function, we make sure that once a solution is found (the goalDistance is zero) the individuals are also evaluated by the number of turns that they have taken to reach the goal. If we keep on generating new generations after the goal is reached by some individuals this means that new generations will continue optimizing the number of turns required to reach the goal.

For efficiency we precalculate the distance from every position of the track to the goal using breadth-first search when a track is loaded. These distances are then put into a distance matrix D, where $D_{x,y}$ is the distance from the point (x, y) to the goal. Using this matrix the fitness function is calculated in constant time.

4.3 Tracks

During the implementation of the algorithm we created several tracks to test different aspects of the agent. Below are the different tracks and the corresponding track specific attributes of our algorithm.

4.3.1 Oval

This was our first and simplest track and just consists of a simple oval.

Table 1: Attributes (the .lvl file) for the oval track.

sensor_near	2
sensor_normal	6
goal_distance_weight	1.8
generations	100

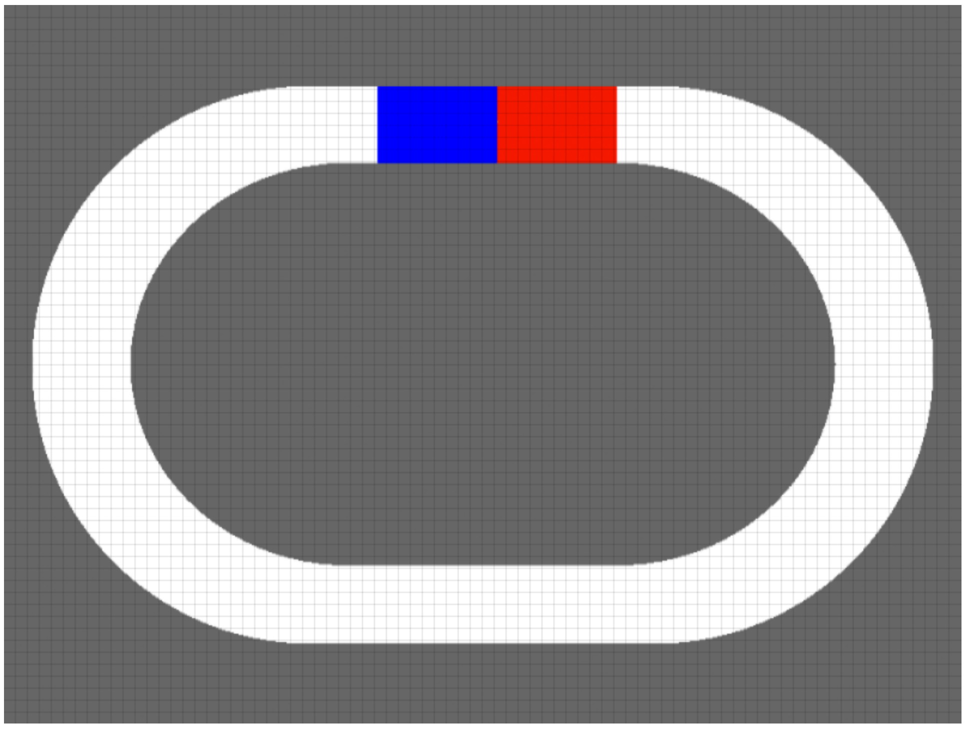

Figure 8: The oval track.

4.3.2 Comb

This track consists of many hairpin turns, and the idea is to challenge the agents ability to take many varying and sharp corners.

Table 2: Attributes (the .lvl file) for the comb track

sensor_near	2
sensor_normal	4
goal_distance_weight	3
generations	1000

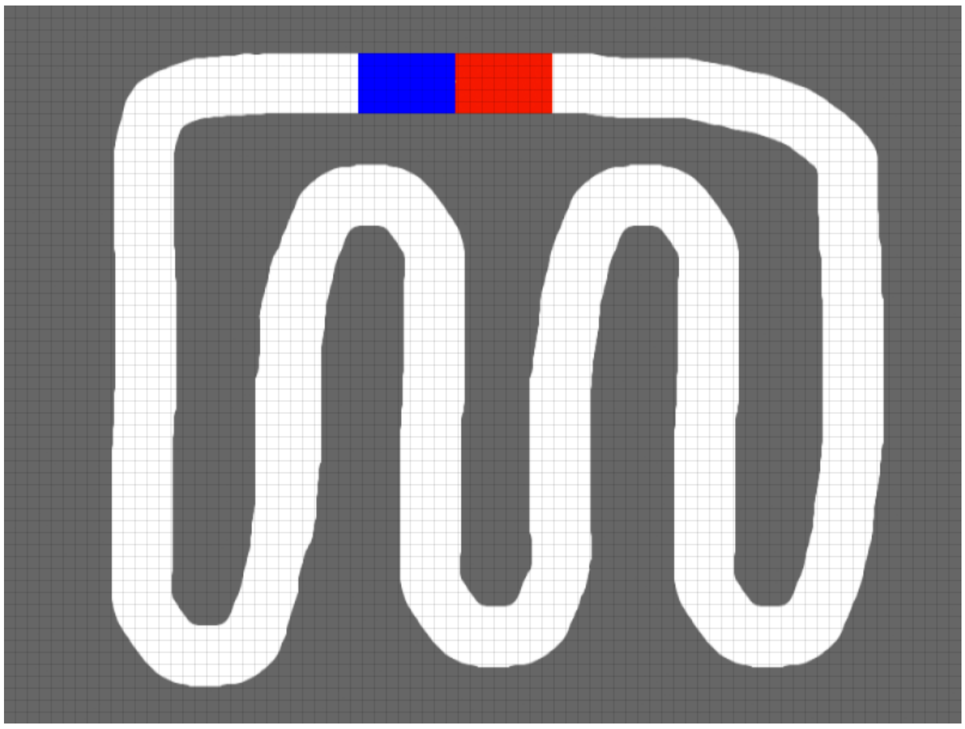

Figure 9: The comb track.

4.3.3 Choices

This track has several different paths from start to goal and its main purpose is to test how the agents reacts when they are faced with forks on the track.

Table 3: Attributes (the .lvl file) for the choices track

sensor_near	2
sensor_normal	6
goal_distance_weight	3
generations	1000

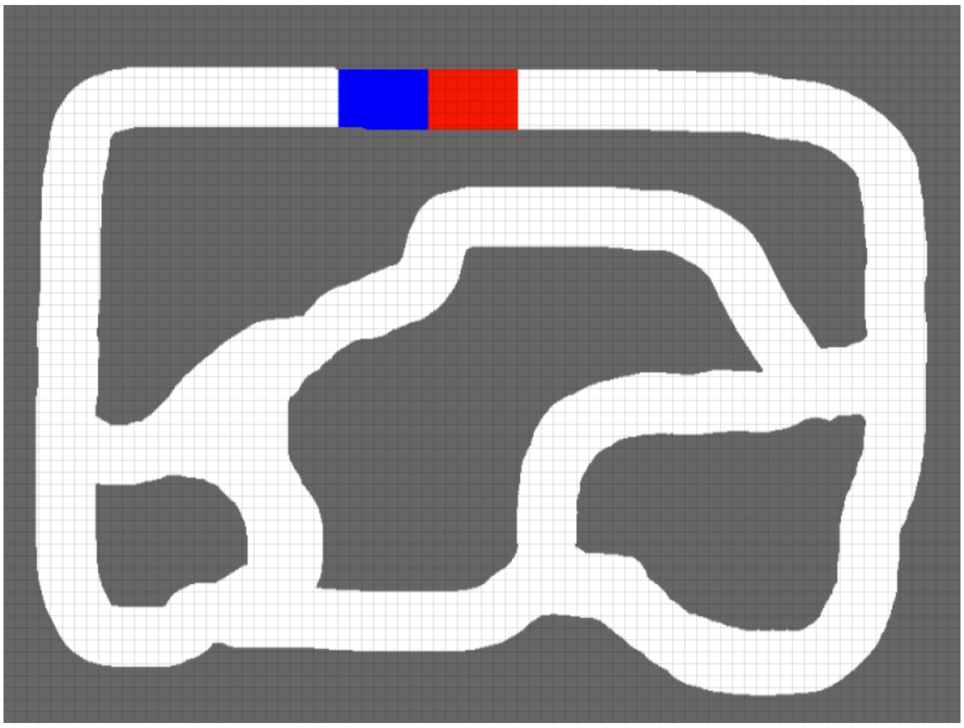

Figure 10: The choices track.

4.4 Optimizations

4.4.1 Genome size

After a time of testing we realized that since the inertia vector used in the genome is absolute relative to the track, the agent treats two identical states different depending on which direction it came from.

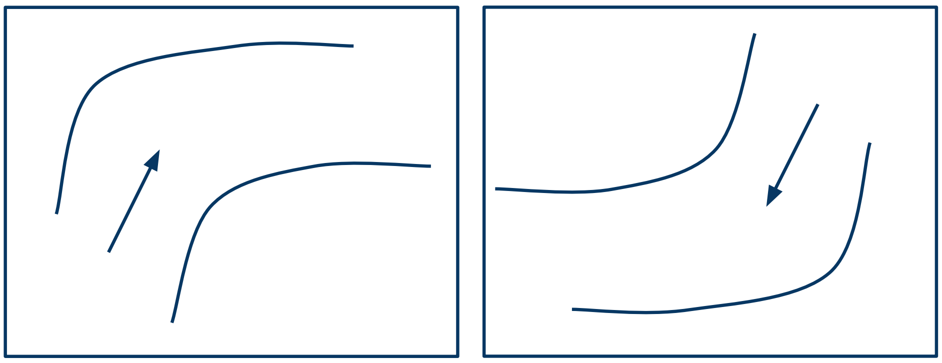

Figure 11: Two different states that are treated as identical states by the optimized algorithm.

Since the decisions and sensors are relative to the agent, the agent should make the same decision in these two states. To do this we rotate the inertia vector to be relative to the agent when deciding which state the agent is in. This reduces the genome size and makes it more relevant.

4.4.2 Better fitness function

During our optimization we also experimented with the fitness function. We added a weight to the goalDistance part of the function (see formula 3).

$$f(agent\_position, num\_turns) =$$

$$goal\_distance\_weight * goalDistance(agent\_position) + num\_turns \quad (3)$$

The *goal_distance_weight* attribute is specified in the .lvl files, and the reason why we vary the weight is because some more advanced levels benefit from a higher goal distance bonus. Other simpler levels could instead benefit if the *num_turns* part of the fitness function was dominating, as the agents will find a way to the goal fast anyway and then could spend time optimizing the number of steps.

4.4.3 Clone top individuals

To speed up the evolutionary progress we clone the selected individuals (the top 10 performing) of each generation and include these clones in the next generation together with their 100 children. This makes sure that the top individuals of each generation are at least as good as the last generation's top individuals. Doing this allows us to have a higher mutation probability, speeding up the genome development and the evolutionary progress. If we have a high mutation probability without cloning the best individuals, too many of the good genes from the best individuals get mutated during the crossover, and individuals do not really evolve from generation to generation.

In nature this type of cloning does not really occur, but instead the mutation probability usually is lower so good genes are passed on to new generations with a higher probability. In our case the cloning approach gave a much better performance as generations evolved much faster with respect to the fitness function.

5 Results

Using our approach we managed to solve all our tracks with a reasonable number of generations. The implementation can be found at
`https://github.com/torandi/racetrack_agent`

5.1 Paths

Below are examples of paths that individuals that reached the goal took. Even though several generations was run after the goal was reached by some generation, these paths are not optimal in any way.

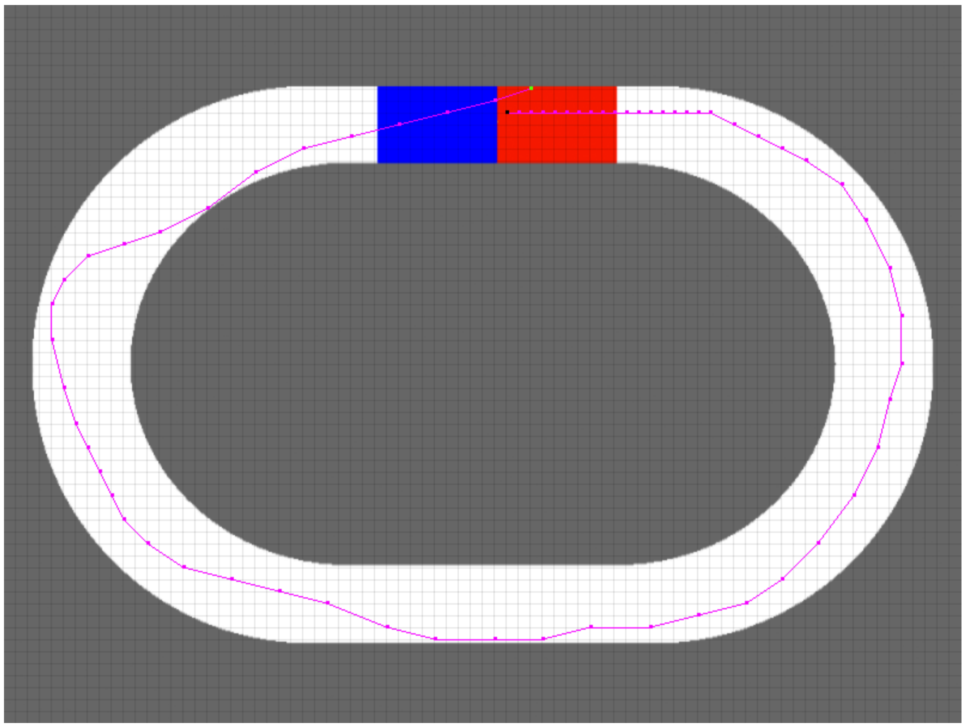

Figure 12: The path of an individual that completed the oval track.

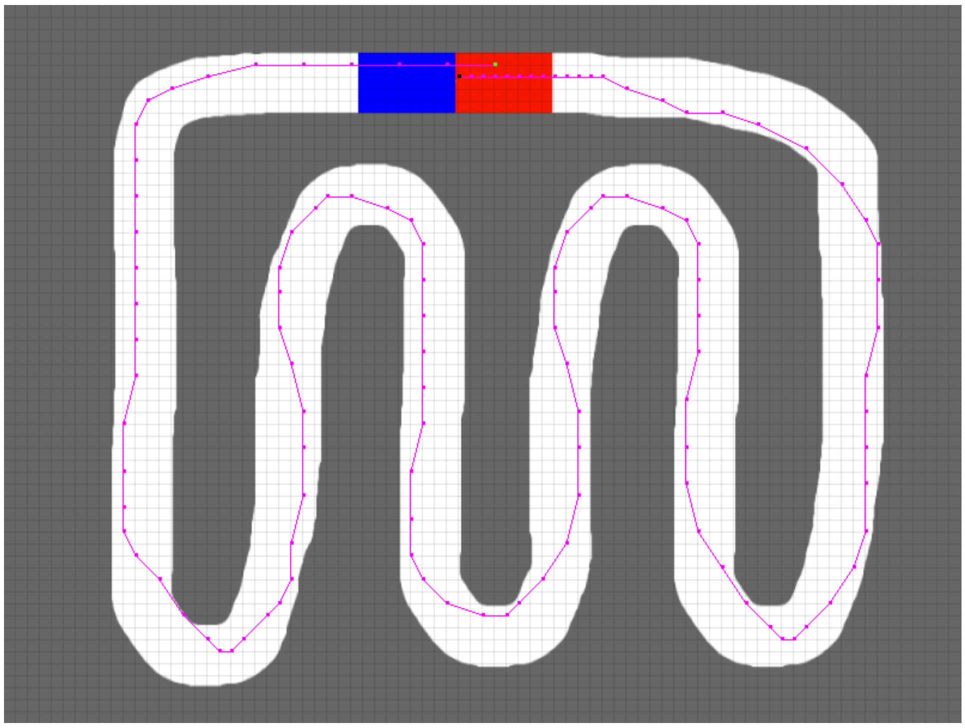

Figure 13: The path of an individual that completed the comb track.

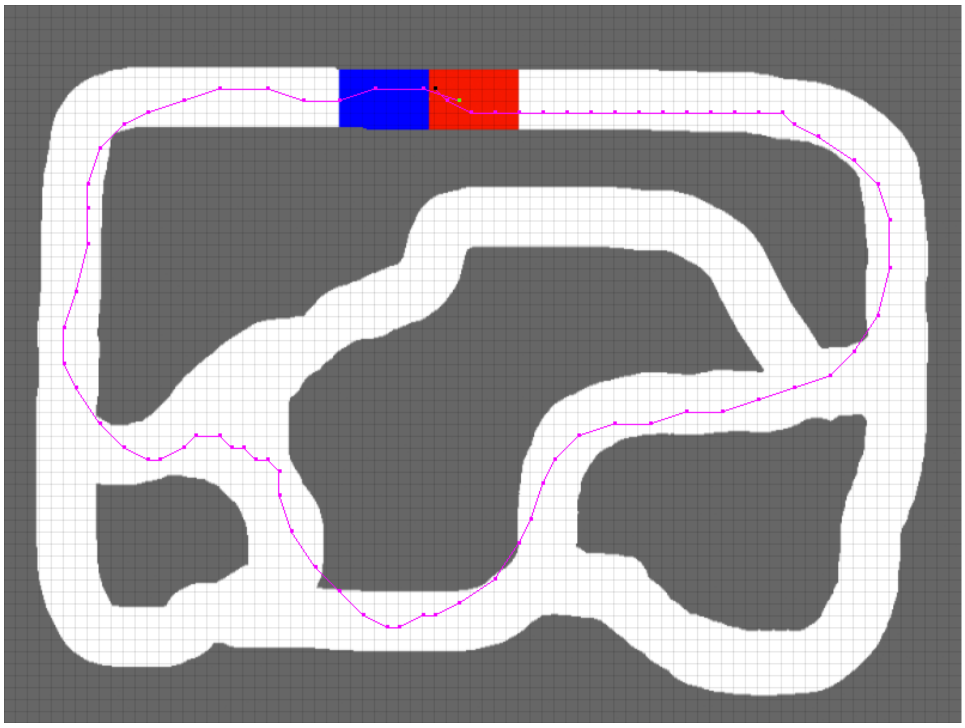

Figure 14: The path of an individual that completed the choices track.

5.2 Performance

5.2.1 Fitness over time

These graphs illustrate the fitness function's value per generation of our final algorithm implementation. Please note that a lower fitness value is better. Each graph shows both the average fitness of an entire generation and the average fitness of the top ten individuals of the generation.

The mutation probability used when nothing else is specified is 0.10.

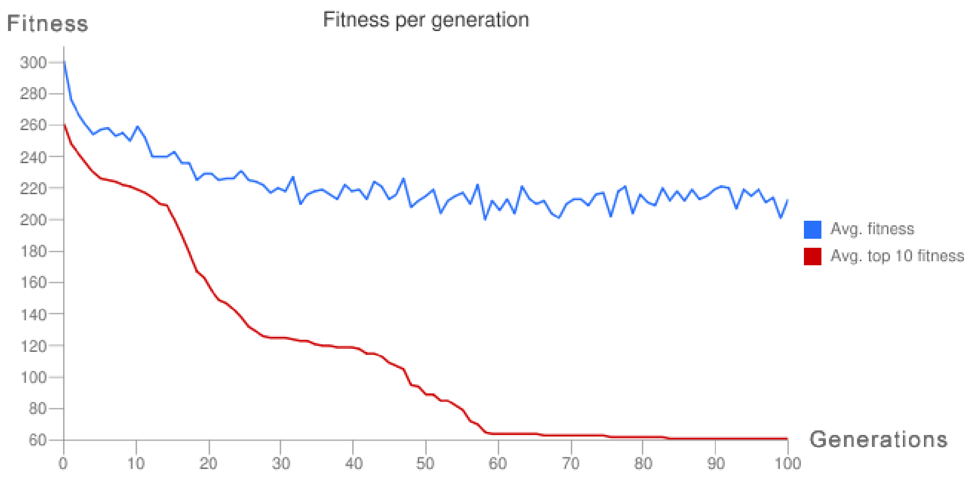

Figure 15: **Oval**: A graph with the fitness per generation.

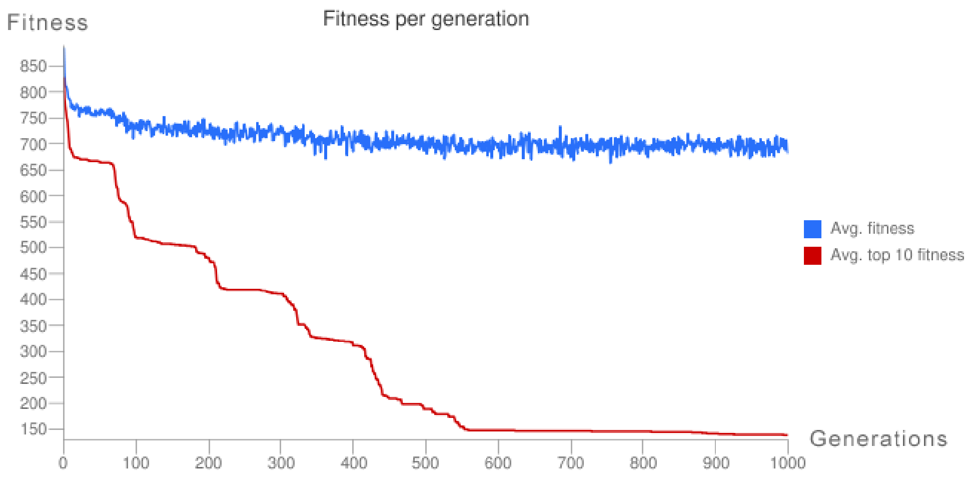

Figure 16: **Comb**: A graph with the fitness per generation

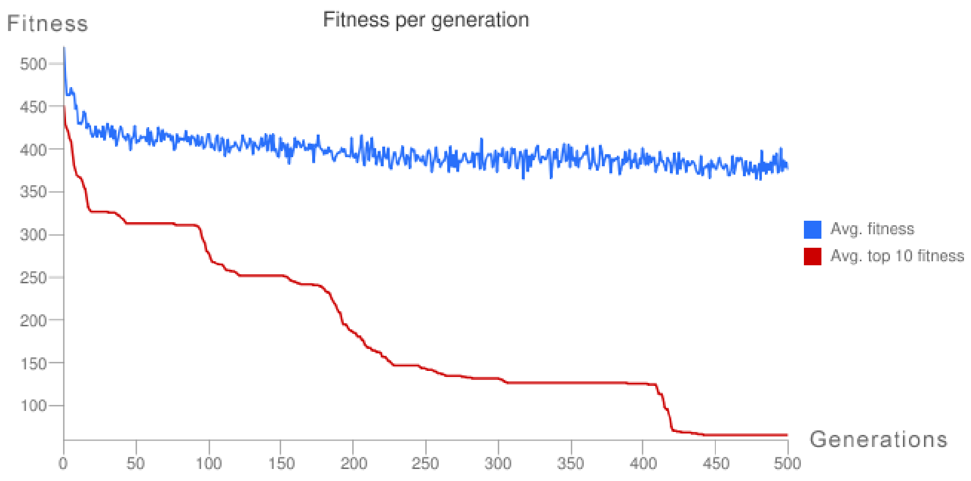

Figure 17: **Choices**: A graph with the fitness per generation

5.2.2 Optimizations

The following graphs show how different optimizations affected the algorithm. The figures 18 and 19 correspond to the optimizations in section 4.4. Figure 20 displays the performance of the algorithm when using different mutation probabilities.

No graph is present for the fitness optimization (section 4.4.2) since this optimization modifies the fitness function itself and therefore can not use the fitness function to measure the relative performance with and without the optimization.

These graphs shows the average fitness value of the top 10 individuals in each generation. The mutation probability used when nothing else is specified is 0.10.

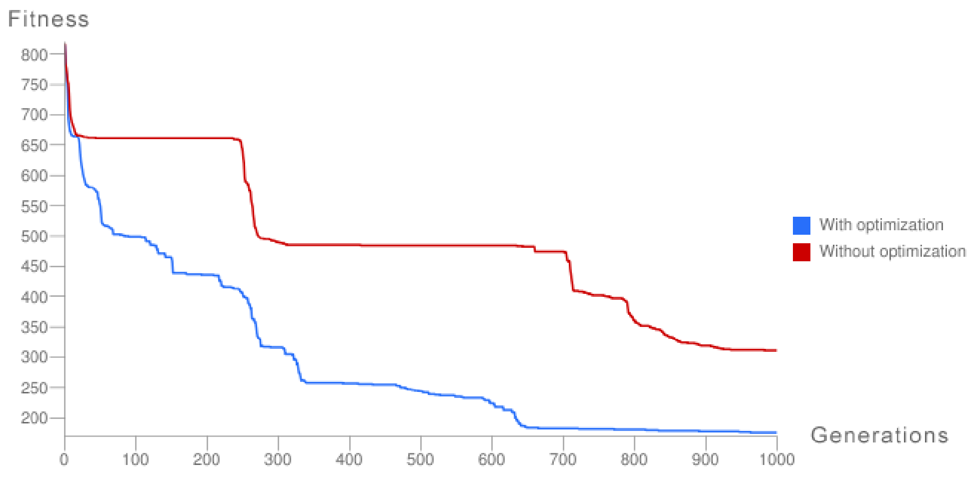

Figure 18: A graph with the average of the top 10 individuals in each generation on the track Comb with and without the genome size optimization (section 4.4.1).

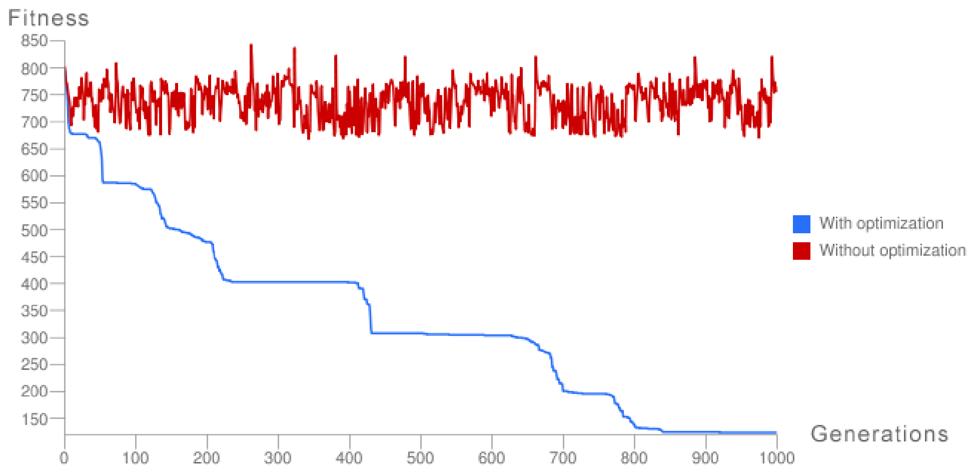

Figure 19: A graph with the average of the top 10 individuals in each generation on the track Comb with and without the cloned parents optimization (section 4.4.3).

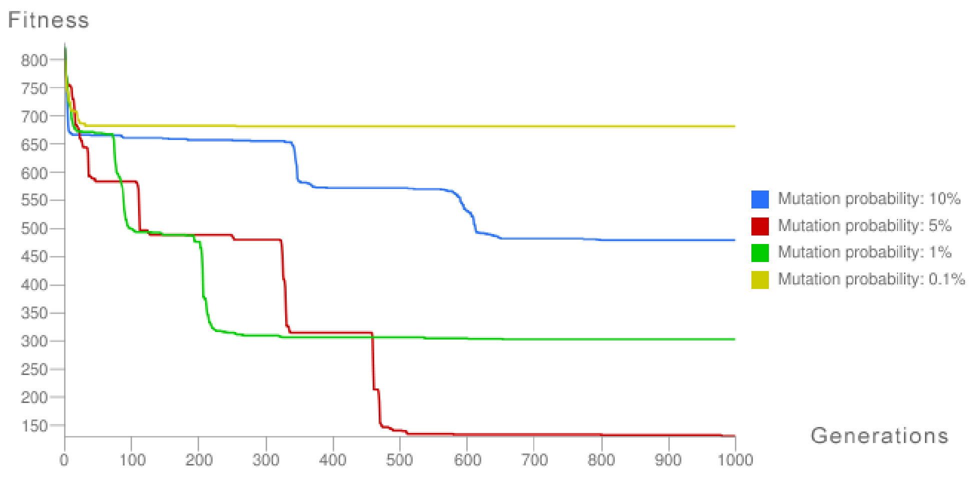

Figure 20: A graph with the average of the top 10 individuals in each generation on the track Comb with different mutation probabilities

6 Discussion

6.1 Results

Oval was the simplest track of the ones that we tested, and was generally solved under 100 generations. This was expected as it only contains very similar right hand turns for the agents, meaning that only parts of the genome is used. States in the genome corresponding to for example sensor values of left hand turns becomes irrelevant and only part of the genome has to evolve. Comb and choices required more generations before individuals reached the goal because of the higher complexity of the tracks, and the need for bigger parts of the genome to evolve.

The genome size optimization makes more parts of the genome reusable, which increases the chance of the agent knowing how to take a curve without crashing. This can be seen in figure 18 where the optimized algorithm reaches a lower fitness value faster.

Cloning the parents into the next generation gives a higher stability, preventing the algorithm from loosing ground. This though slightly increases the risk for the algorithm to get stuck, which occurs if the top 10 individuals (the parents) do not produce children that get better fitness values than themselves. That will result in the same parents' clones getting passed on for generations. As can be seen in figure 19 the algorithm is very unstable without the optimization though (much due to the high mutation probability of 0.1).

From figure 20 we can see that a mutation probability of 0.1 might not be optimal (this is the value that have been used in all test except in the mutation probability test). According to figure 20 a mutation probability between 0.01 and 0.05 seems optimal.

In our implementation there are some track specific variables; the sensor distances and the weight in the fitness function (see 4.4.2). The sensor distances can not be the same on a track with broad roads as on a track with small narrow roads. The fitness weight could have a high static value (prefer lesser goal distance), but this would give us a less optimal result (in terms of number of moves for the agent to reach goal). To have a lower value would make it less likely for the agents to finish the more difficult tracks.

Some of the track specific attributes, like the sensor settings could probably be integrated in the genome to make the algorithm able to solve more generic tracks without human input. The increase of the genome size this would cause is negligible.

6.2 Reliability

Since the algorithm contains stochastic elements the results vary between each run. We have tried to pick statistics from runs that are average when generating the graphs. The same applies to the optimizations; even though the two compared runs uses the same seed the given seed could give an unfair advantage to one of the methods. To prevent this from affecting the result too much we have run the tests multiple times to make sure the tendency is correct.

6.3 Applications

Our experiments have shown us that it is possible to use genetic algorithms in games. Using genetic algorithms would create an opponent that learns from the environment and becomes better over time, but also gives us an agent that does not behave deterministic. Other positive effects could be an agent that learns from the player's behavior, which would encourage the player to change strategy when the opposition becomes to difficult.

The downside to genetic algorithms is that it can be difficult to express a problem as a genetic algorithm. One must be able to describe an entire individual as a fix sized genome. Another downside might be the performance as the evolutionary progress can be slow when using big genomes. In our problem we have quite a simple and limited game making the genomes relatively small. In nature, evolution really is a very slow process that can take millions of years to adapt a species to new environments.

References

[1] Racetrack (game). `http://en.wikipedia.org/wiki/Racetrack\_(game)`, 11 april 2011.

[2] Stuart Russel and Perer Norvig. *Aritificial Intelligence - A modern approach.* Pearson, 3:rd edition, 2010.