Machine Learning – Project

Problem description

The learning task consists in finding the optimal ordered sequence of actions that a car should do for completing a lap of a specific track using genetic algorithm.
The environment is composed by a bitmap image representing the track and the population is composed by a finite number of cars.
A 2D engine, that checks for inconsistencies with the environment and modifies the cars' statuses according to the environment's reaction, manages all the actions made by the cars.
At each clock's tick, representing the passage of time, each car, interfacing with the 2D engine, does a move.
After a finite number of ticks the genetic algorithm will select the fittest cars for composing the new generation that will restart the race.
This makes the fittest cars' DNA converging to an optimal sequence of actions for cutting the finish line.
Since the genetic algorithms tends to maximizes a given fitness function and since this fitness strongly depends on the environment, once the genetic algorithm is defined, in terms of DNA and operators/mutations, the optimality of the solution depends only on the goodness of the environment.

Luca Mastrostefano 1569186

Environment

Environment

Each car has its own status that includes:

- Current position: a pair of float numbers that represent the its position on the map;
- Speed: a float number representing the speed as pixels/move;
- Steering wheel angle: the angle in radiant between the horizontal axis of the track and the car's direction;
- Number of crashes: an integer number increased each time the car hits an obstacles or an edge of the track;
- A boolean value: representing if the car has finished the track or is still running;
- Number of turns made: an integer number increased each time the car moves;
- Lot of other variables…

The actions that a car can perform are the following:

- BREAK => $speed \leftarrow \max(0, speed - c_{speed})$
- ACCELERATE => $speed \leftarrow speed + c_{speed}$
- ROTATE_LEFT => $theta \leftarrow \left(theta + \frac{c_{theta}}{1+speed}\right) \%\ 2\pi$
- ROTATE_RIGHT=> $theta \leftarrow \left(theta - \frac{c_{theta}}{1+speed}\right) \%\ 2\pi$
- DO_NOTHING

Notice that c_{speed} and c_{theta} are two positive constant and that dividing theta by the speed results in turning less as speed increases.

The track is a bitmap image with edges and obstacles. To make these elements visible to the 2D engine they have to be black (#000000):

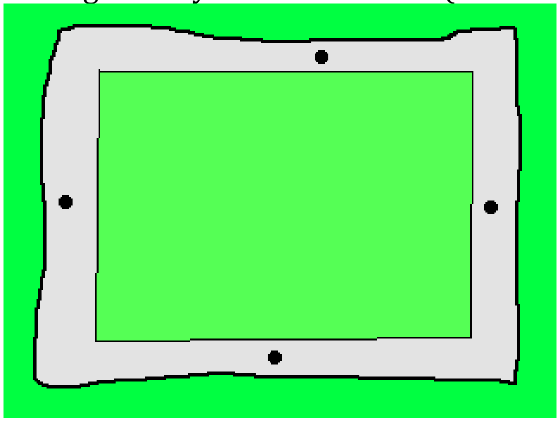


Figure 1 - Example of a track with four obstacles inside.

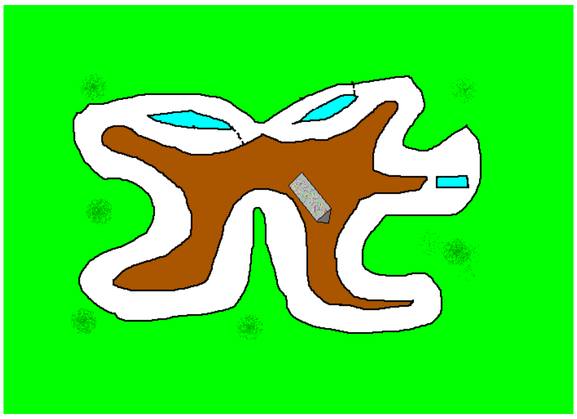


Figure 2 - Example of difficult track

The black pixels are not traversable and, if the car were to crash against it, it will be stopped (the speed of the car is set to 0 and the number of crashes is increased).
Once the image is loaded the user can draw the start line on the track:

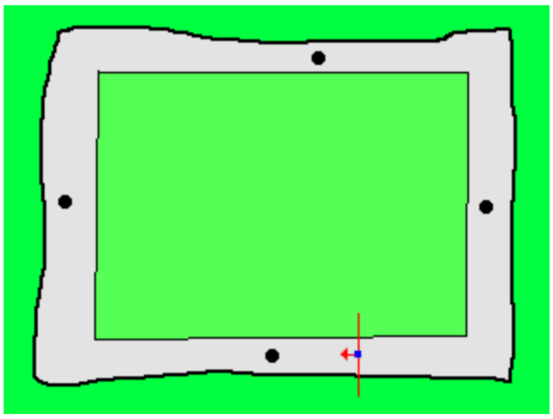


Figure 3 - Example of correct start line

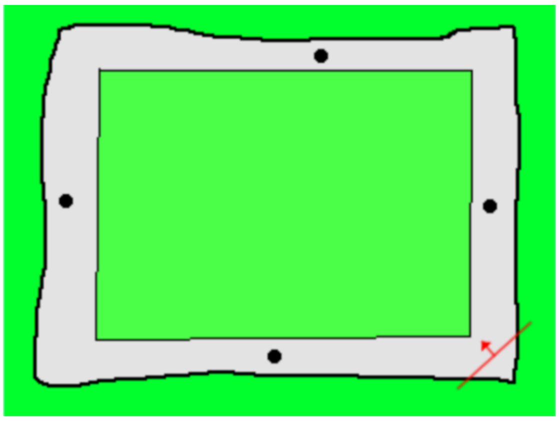
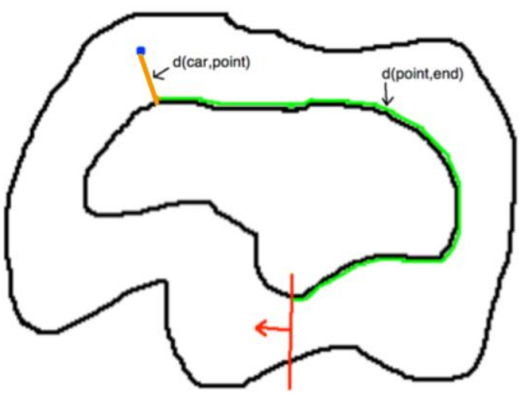


Figure 4 - Example of a wrong start line

During the drawing of the start line the system detects the number of black pixels on which such start line passes and checks if those black pixels belong to two different lines. If these lines were not two distinct lines the start line is rejected, otherwise the shortest line will be stored as the internal edge of the track.
Once the start line has been found, each pixel of the internal line is explored with a directional-bfs starting from the intersection between the internal line and the start line (each pixel is added to the queue of the pixel to visit only if visiting it from the current pixel does not imply crossing the start line conversely). This operation allows the system knowing the distance from each point of the internal edge to the end of the track and those distances will be useful for computing the fitness later.
To compute the distance from the end of a specific car we can leverage on the fact that each point of the internal edge of the track knows its own distance from the end. To approximate the distance from the end we can sum the distance from the end of the nearest point of the internal edge of the track to the car and the distance of the car to this point:



During each turn the 2D engine computes the next position of all the cars according to their speed and wheel angle, detecting any collision either with edges or obstacles and check if any car has crossed the finish line in the right direction.
This particular operation is repeated exactly $generationNeeded * populationSize * numberOfActions$ times.
Since we could need hundreds of generation, hundreds of member and thousands of actions to compute an optimal solution, that system will compute tens of millions of moves.

Moreover, every time a generation ends the system has to compute the fitness for each car and apply all the mutations needed for the creation of the new generation.
Since we have to compute the fitness of all the cars every time a generation ends, the system will compute $generationNeeded * populationSize$ (about 10-100K times) the fitness and all the mutations.

Thanks to the Bresenham algorithm[1] and lot of optimizations and thanks to the precomputed distance from each point of the internal edge to the end of the track, the system is able to compute 1-2 millions of moves and fitness every second, making the algorithm ends in tens of seconds also for long tracks.

Fitness functions

Evolution of fitness function

Lot of fitness functions have been studied in order to find the one that is higher for cars that run more and are more efficient, minimizing the time needed to complete the track.
The first and simpler fitness function studied was the one that gives higher values to cars that maximize the distance from the start:

$$fitness = distanceFromStart$$

But this could not imply minimizing the distance from the end:

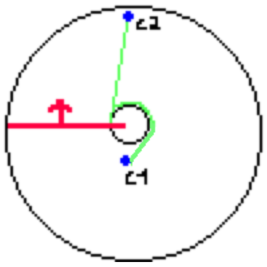


Figura 5 - d_start(c1)<d_start(c2) and d_end(c1)<d_end(c2)

In fact, even if car c1 is more close to the end, it is also more close to the start.
This simple reasoning leads me in discarding such fitness function and starting minimizing the distance from the end:

$$fitness = \frac{1}{(1 + distanceFromEnd)}$$

Once understand that the distance to use is the distance from the end, I have tried to add a component that gives higher value for car that minimize the number of moves:

$$fitness = \frac{1}{(1 + g(distanceFromEnd, numberOfMoves))}$$

Depending on function g the fitness can change a lot, since the distance from the end decreases with the passage of time and the number of moves increases with the passage of time.
In fact, if

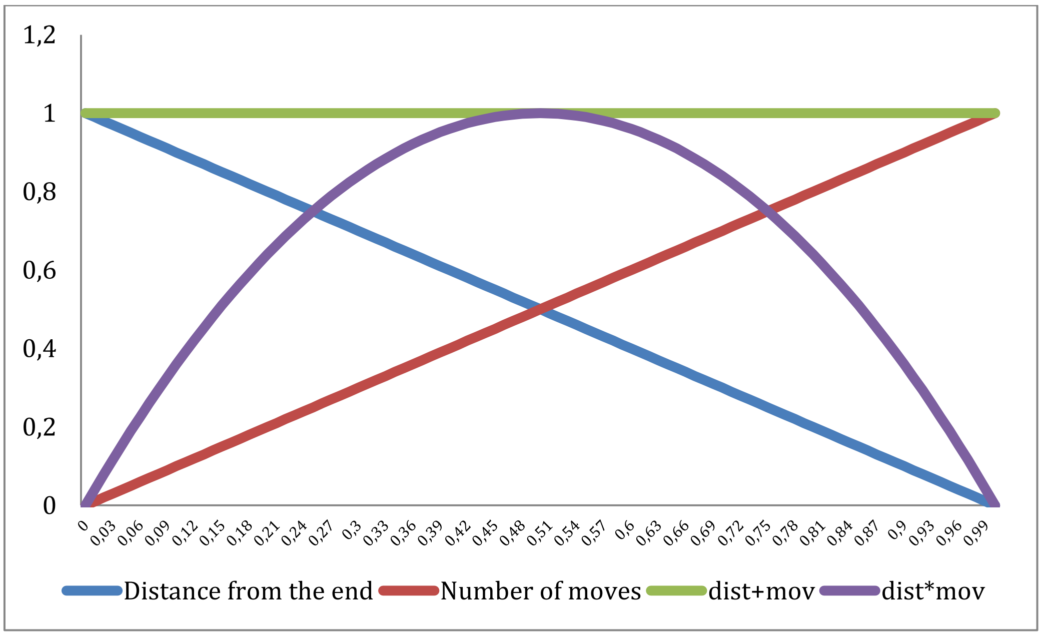
$$g(distanceToEnd, numberOfMoves) = distanceFromEnd + numberOfMoves$$

[1] http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

the resulting fitness function would have been almost a constant.
Otherwise if

$$g(distanceToEnd, numberOfMoves) = distanceFromEnd * numberOfMoves$$

the resulting fitness function would have been a convex parabola, making very difficult the scouting of better hypotheses when $\frac{\delta(fitness)}{\delta t} < 0$.
In the graph below it is plotted the denominator of the fitness function depending on function g:



But, since in a given turn the car that minimize the distance from the end is also the one that has done the most effective moves, a good fitness function is the simplest one:

$$fitness = \frac{1}{(1 + distanceFromEnd)}$$

With this fitness function we could find a very good solution, cars run a lot and they can easily reach the end. But they are not so motivated to avoid crashes.
In fact, what happens during the straight parts of the track is that the cars prefers to run as much as possible and then use the internal edge of the track to break, because the crash will make them break immediately.
Therefore, to penalize more cars that crash against the obstacles, we can introduce an additional penalty in the fitness function:

$$fitness = \frac{1}{(1 + distanceFromEnd \; + \#crashes * c)}, \qquad c \gg 1$$

The above fitness function performs very well: now cars avoid crashing and they break before a curve.
But once the cars reach the end, avoiding crashes, their fitness is equal to:

$$fitness = \frac{1}{(1+0\ \ +0*c)} = 1$$

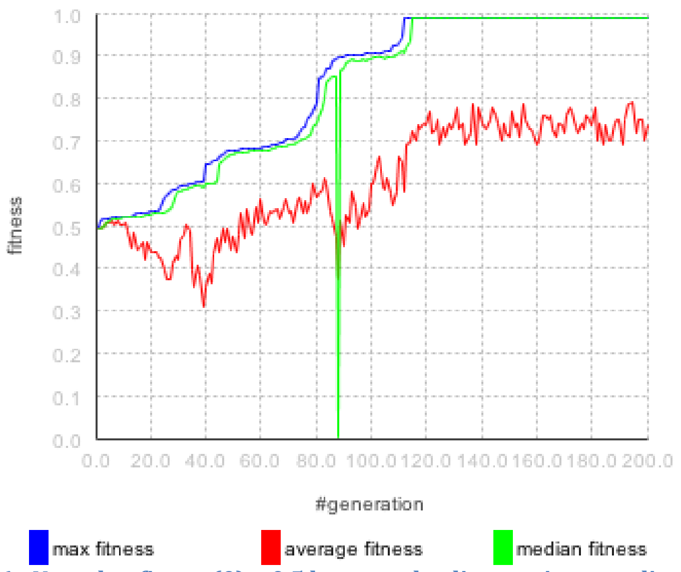


Figura 6 - Note that fitness(0) = 0.5 because the distance is normalized and f(0) = 1/(1+1)

But being equal for all the cars (median fitness= max fitness) after generation 110, we can't continue learning!

To continue optimizing the solution found, also after that the cars reach the end, we can split the fitness function and start using a different function for cars that have crossed the finish line:

$$fitness = \begin{cases} 0.5 + \dfrac{0.5 * e^{-\#moves}}{(1 + \#crashes * c)}, & end\ reached \\ \dfrac{0.5 * e^{-distanceFromEnd}}{(1 + \#crashes * c)}, & otherwise \end{cases}$$

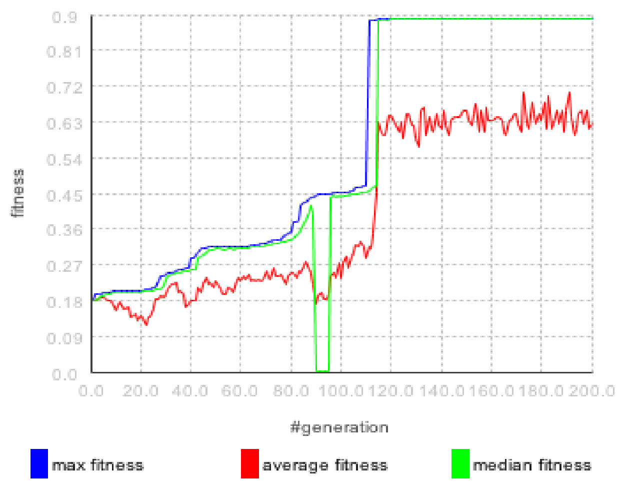


Figura 7 - Note that fitness(0) = 0.18.. because distance is normalized and f(0) = 0.5*e^(-1)

The big gap between 0.5 and 0.88 at generation 110 is due of the discontinuity of the fitness function, but if we zoom on generation 110, we can see that the cars are continuing to learn:

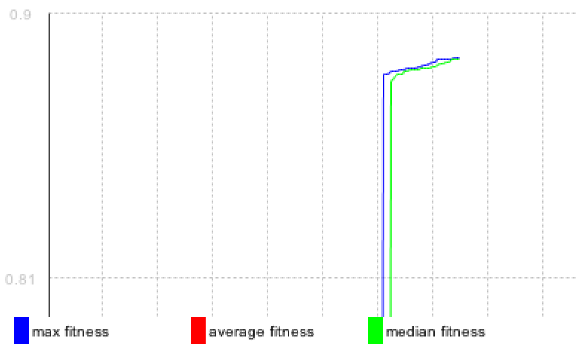


Figura 8 – Cars continue learning also after they reach the end

As we can see from the above graph, now cars are continuing learning but what will happen is that they will learn just a little bit, because the time is discrete and the smallest unit for measuring their fitness is a move:

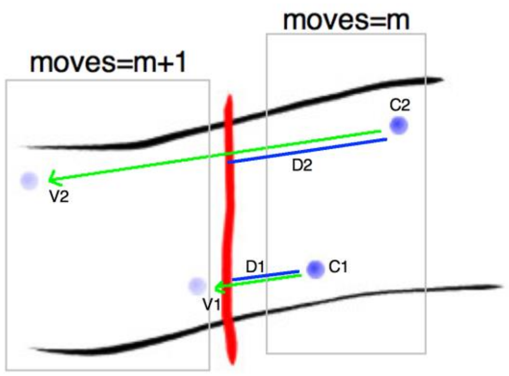


Figura 9 - Two cars cutting the finish line with the same number of moves, but still we can define who is better than the other.

Since the fitness function is depending only on the number of moves, if two cars cut the finish line with the same number of moves, as shown in the above image, their fitness will be the same.
To avoid this and to continue learning we can extend the concept of a "move" just for the final one and taking in account also the time needed for cutting the finish line:
In the previous example #moves of car C2 becomes:

$$moves = \#moves + sig(D2/V2)$$

and the one of car C1 becomes:

$$moves = \#moves + sig(D1/V1)$$

where $sig(x) = \frac{1}{1-e^{-x}}$ and, being distances and speeds positive:

$$0.5 < sig(t) < 1, \forall\, t = D/V$$

Adding to the number of moves such component, we are considering the time no more discrete but continuous when a car cuts the finish line.
Note that since $sig(t)$ is always in range [0.5,1), number of moves remain the dominant factor in computing the fitness.

<u>How to make fitness comparable between different tracks?</u>
Since it would be better having comparable fitness function between tracks, the number of moves and the distance from the end were normalized using a precomputed length of the track:

$$normalizedDistanceFromEnd = \frac{currenDistanceFromEnd}{internalEdgeLength}$$

$$normalizedNumberOfMoves = \frac{currenNumberOfMoves}{internalEdgeLength}$$

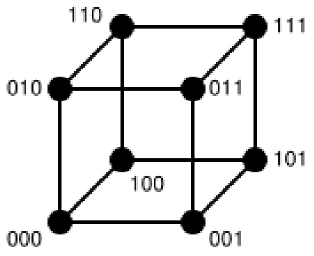
With such normalizations $normalizedDistanceFromEnd$ represents the percentage of remaining road to be covered and $normalizedNumberOfMoves$ represents how many moves are needed to pass through a pixel in average.
Moreover, other constants have been summed or multiplied to make the fitness function vary between [0,1].

Genetic algorithm

DNA representation

The DNA is represented by an ordered sequence of actions. The action is the smallest unit composing the DNA and is encoded with a unique integer ID:
0 = BREAK
1 = ACCELERATE
2 = ROTATE_LEFT
3 = ROTATE_RIGHT
4 = DO_NOTHING
Therefore the DNA is made up of a sequence of integers and the position of such integer represents the number of turn in which the relative action will be performed:
DNA: 041412322103131333O1...

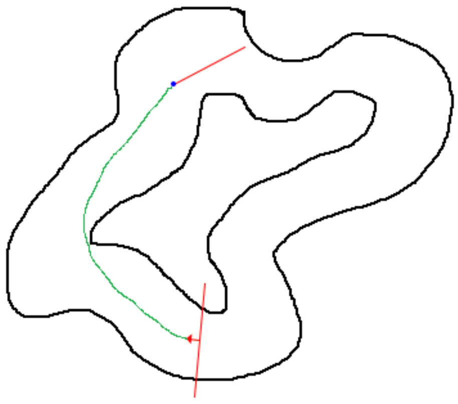
Notice that the DNA is not a bit string, because, being so, it would have existed a distance between actions.
In fact whether any action has been encoded using a binary representation, it would have been require 3 bits:
000 = BREAK
001 = ACCELERATE
010 = ROTATE_LEFT
011 = ROTATE_RIGHT
100 = DO_NOTHING

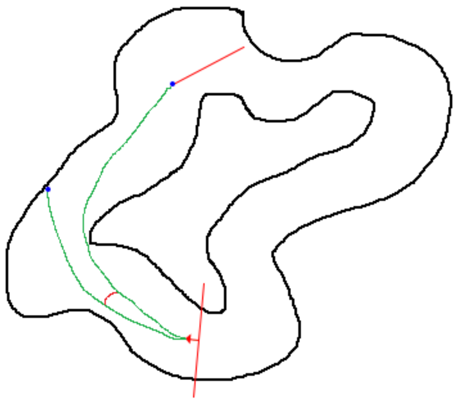


Thus, a random point mutation changes an action with a random action between its nearest (according to the hamming distance), making the random mutation not so random!
Indeed, the hamming distance between 6 actions in a decimal system is equal for any pair of actions and equal to 1.
In general it should be used an n-ary system to perfectly encode n actions.

Operators

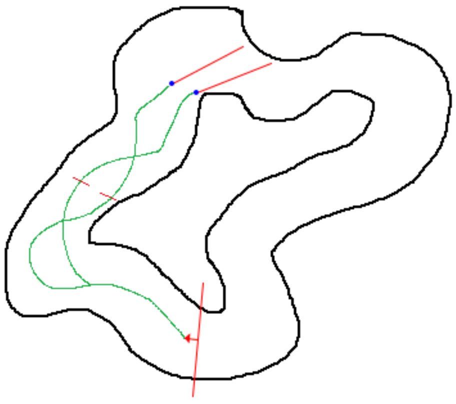
Since the DNA is a sequence of actions ordered chronologically and the status of a car strongly depends on its previous status, a small random change at the very start of the DNA can really affects the car's fitness. Think about changing an action with a ROTATE_LEFT when the speed is not so high: this will result in a significant rotation of the whole track path of the car:

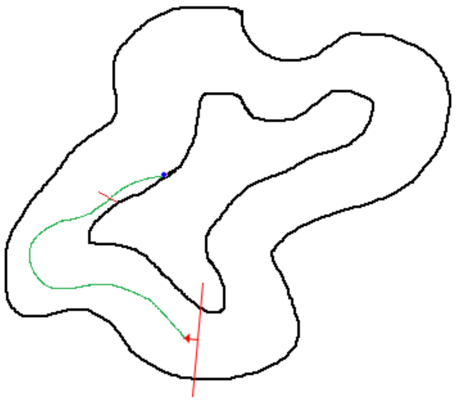




Luca Mastrostefano 1569186

This DNA's sensitivity makes it really instable in case of random changes and even more in case of crossover application:

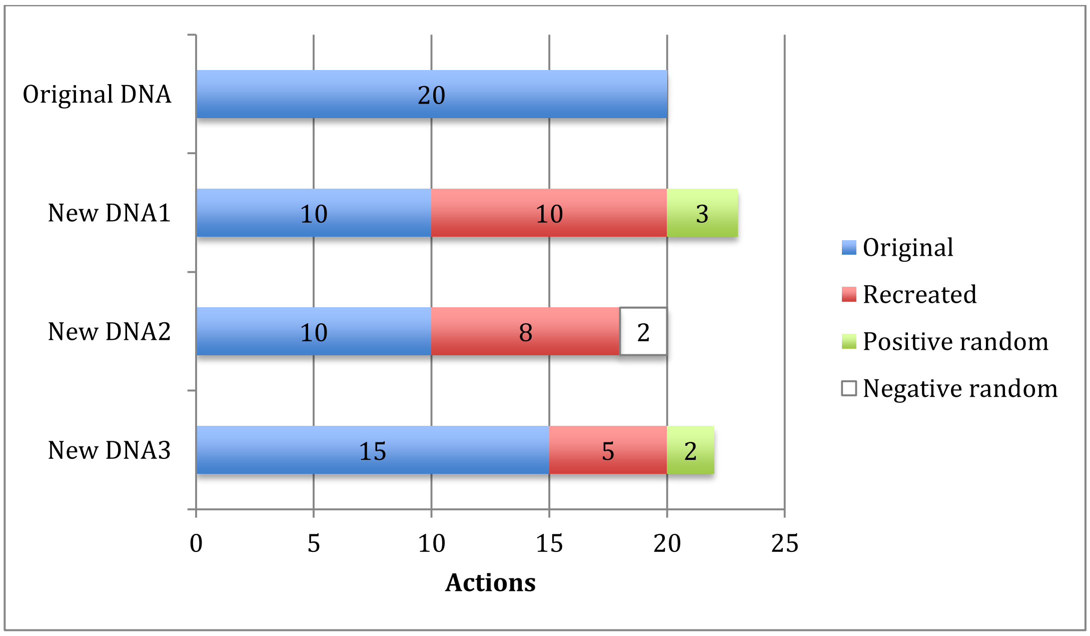




Variable length hypothesis

The intuition that unlocked the situation is due to a research on the "lengthening of telomeres[2]":such event indicates the reproduction of DNA's extremes, otherwise lost during the duplication.

With the same logic, the ending part of the DNA is cut and a new part of DNA will replace it. This new part of DNA has the same length of the removed one plus a random constant that is positive with a probability > 0.5. In this way, the DNA tends to grow slowly.



As the above graph shows, in this example the original DNA was composed by 20 actions and two extremes-DNA-cutter-mutation have been adopted:

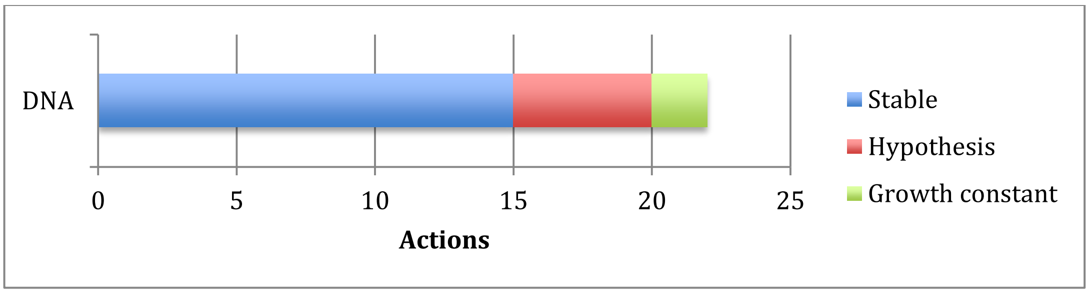
1. cut 10, recreate 10 ± x, with x ∈[-5, 5];
2. cut 5, recreate 5 ± x, with x ∈ [-5, 5] ;

[2] http://en.wikipedia.org/wiki/Telomere

The new DNA have been created as follow:

- New DNA1 has been create using the cutter #1:
  The last 10 actions of the original DNA have been replaced with a random DNA of length 10 and another part of DNA of length 3 (a random number in [-5,5] with higher probability for positive number) has been added to the end of the new DNA.
- New DNA2 has been create using the cutter #1:
  The last 10 actions of the original DNA have been replaced with a random DNA of length 10 and the last 2 actions (a random number in [-5,5] with higher probability for positive number) have been removed by the end of the new DNA.
- New DNA3 has been create using the cutter #2:
  The last 5 actions of the original DNA have been replaced with a random DNA of length 5 and another part of DNA of length 2 (a random number in [-5,5] with higher probability for positive number) has been added to the end of the new DNA.

So the DNA is composed by three parts:



1. The stable part changes very rarely;
2. The hypothesized part, being random, it is almost guaranteed to change from car to car and is an attempt to find a solution to the next part of the track;
3. The growth constant, with the probability of being positive, makes stable a random number of starting actions of the hypothesized part; otherwise, with the probability of being a negative number, marks as not stable a random number of ending actions of the stable part that, therefore, will be probably changed in the next turn.
   In general, with the probability of being a positive number the DNA grows, with the one of being a negative number the DNA reduces itself.

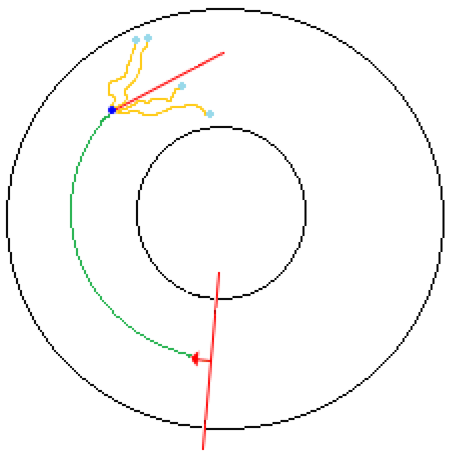
<u>Note that the smaller the hypothesized part the more cars will be greedy, making them less reactive. The greater the hypothesized part the more cars will tend to move away from the current position, trying to get out of local minima and looking for solutions that can temporary bring fitness to drop.</u>

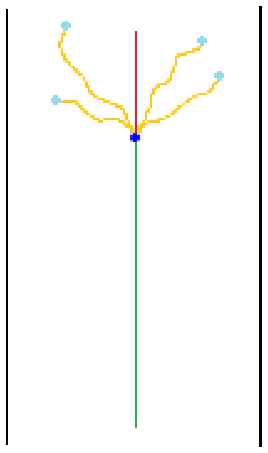
<u>*Which is the best length?*</u>

- If only short-length hypotheses are done, the cars tend to be too much greedy and will start turning just in front of an obstacle, without trying to anticipate slightly the curve and, after a long straight road, it is possible that the only solution with those few actions is to slam against the obstacles.
- If only long-length hypotheses are done, cars accelerate with great suspicion, because the more it is hypothesized the more is likely that the resulting cars will crash (being the hypothesis totally random). Will result to have a greater fitness only those cars that, proceeding slowly, do not crash before the end of the round, storing in the DNA that the best thing is to go very slowly!

Therefore, is strongly recommended to use more than one extremes-DNA-cutter-mutation with different length.

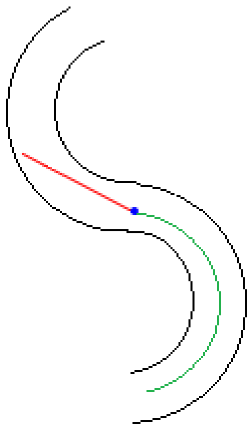
What if we can add inertia to hypotheses?
Why if found a good hypothesis for the generation t, the generation t+1 can't leverage on it, but has to formulate a completely new random hypothesis?





Suppose a set of cars learning how to run on a straight road: cars that do not rotate or break will obtain a higher fitness, because they tend to accelerate most of the time.
Let's focus on the fittest car, whose last actions were most likely ACCELERATE. Now its DNA has been chosen to create new members that will add new completely random actions at the end of the DNA of the fittest car.
Being their hypotheses completely random, it is very rare that a new sequence of ACCELERATE will come up, making their fitness increase with a lower speed and adding some noise in their genetic heritage.
But we can avoid this behaviour, making our cars converge faster and with a cleaner DNA, just picking the actions according to a probability distribution build on the last actions of each car.
Surely we have to protect us against the case in which some action has not been chosen for a while, making their probability to be chosen by the next generation equals to 0.
To avoid such case we can add the Laplace Smoothing[3] to avoid probabilities to be equal to 0.
This reasoning works in the constant parts of the track, but what happens in case of fast changes of the road is that the inertia will make the cars slam.
But we can simply keep part of the populations growing randomly (with a low probability) for being reactive in case of fast changes:

[3] http://en.wikipedia.org/wiki/Additive_smoothing



Adding this inertia to the hypotheses has resulted in number of moves needed to complete a track almost 10% lesser than before and, being more efficient, also the number of generations needed fells by 8-20%, depending on the track's shape.
The 10% gain on the number of moves has been computed as follow:

$$gain' = \frac{\#movesNeeded_{randomCar} - \#movesNeeded_{statisticalCar}}{\#movesNeeded_{randomCar}}$$

but, of course, every track has its own minimum number of moves to be completed, that is not much smaller than $\#movesNeeded_{randomCar}$, but it is unknown.
So the gain should be computed as:

$$gain" = \frac{\#movesNeeded_{randomCar} - \#movesNeeded_{statisticalCar}}{\#movesNeeded_{randomCar} - minNumberOfMoves}$$

Obviously the $gain" \geq gain'$, and, being the solution of the random car already sufficiently good, follows that $gain" \gg gain' = 10\%$.

Light mutations
Since previous mutation affects just the end of a DNA, other mutations have been added:

- Mixer: with a fixed probability a certain number of consecutive actions are mixed:
  Eg: 01042123 -> 01042212 3 or 01042123221241 -> 01041223221421
  This kind of random mutation is really light, in fact it does not introduce a random change, indeed, its effect is trying to anticipate a curve (bringing chronologically back a ROTATE_X action) or trying to make the car faster when leaves a curve (bringing chronologically back an ACCELERATE action);
- Switcher: with a fixed probability an action is switched with its opposite;
- Random changer: with a fixed probability an action is changed with a completely random action.

How to select member for the next generation?
As pointed out above, the stable part of the DNA changes very rarely, so it is very important to choose carefully which members will compose the stable part of the DNA.
This need led us discarding the probabilistically selection of members according to their fitness and composing the next generation by choosing a fixed ratio of population that maximize the fitness function.

So, instead of growing slowly looking for a statistically good solution with the probabilistically selection, with the fittest-selection the algorithm converges faster and we can continue looking for other solutions just restarting it more times.
Altering the ratio of survival members will affect a lot the convergence speed, making it very high in case of a low ratio, but increasing the possibility of stacking in local minima.

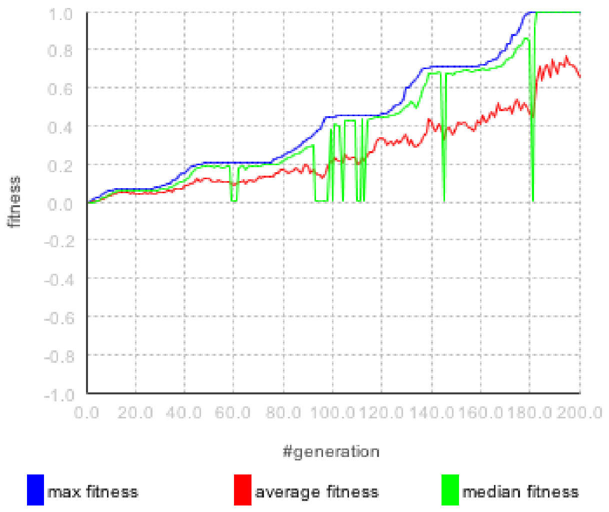
Best operators combination
The analysis of all this operators led in finding a combination of them that works well in almost every track:

```
Starting with a population of 100 members do:
    While (not track terminated) do:
        Execute DNA of all the cars;
        Evaluate cars' fitness;
        Sort them according their fitness;
        Create two new empty sets of members: newGen, immutableGen;
        Select the best 40% of the population from the current generation;//selection
        Get best 5 members and add them to immutableGen;  //prevention
        Duplicate these 40% to obtain again |population| - 5 members   //duplication
        Add these 95 members to newGen;
        Take 20% of newGen and mutate 1 random action;   //random mutation
        Take 20% of newGen and reverse 1 random action;  //reverse
        Take 20% of newGen and switch 2 random action;    //switch
        Take 50% of newGen and cut/recreate 5 ± x, with x ∈[0, 5];  //short-term hp.
        Take 50% of newGen and cut/recreate 10±x, with x ∈[0, 5]; //medium-term hp.
        Take 50% of newGen and cut/recreate 25 ± x, with x ∈[0, 5];  //long-term hp.
        Current generation ← newGen ∪ immutableGen
```

Results

Local minima and turning difficulty

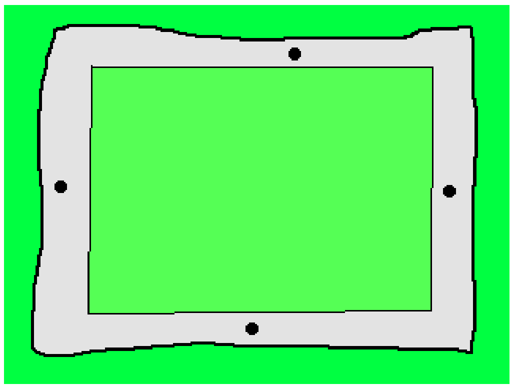
As you can see in the example of fitness function shown in the graph below, in this particular track the derivative of the fitness function on the generation is almost 0 in 5 well-defined ranges:
[10-28], [48-75], [95-120], [140-160], [180-200]



It is simple to understand that, since the fitness is 1, the last range [180-200] is because the cars have reached the end.
The other ranges, indeed, identify that the cars are trying to turn a curve: during a curve, their speed decrease and the distance from the end does not decrease fast as during a straight road.
In fact, being the fitness function ∝ −distanceFromEnd:

$$\frac{\delta(distanceFromEnd)}{\delta generation} \cong 0 => \frac{\delta(fitness)}{\delta generation} \cong 0$$

Therefore, the fitness function above highlights four curves in the track and in fact the track was the following:

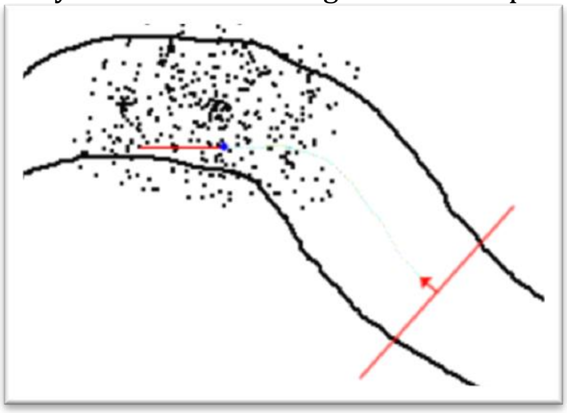


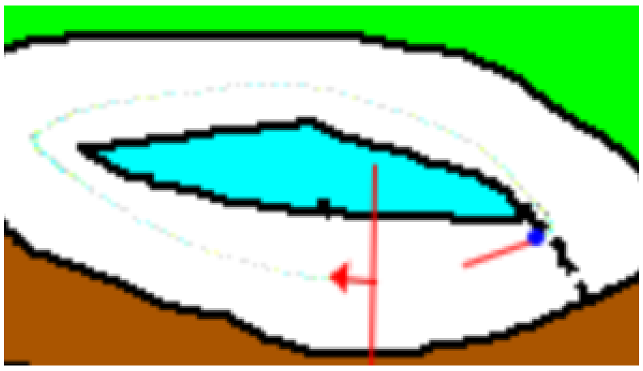
In general the situation in which the derivative of the fitness function is near to 0 denotes a difficulty in learning. Such situation can be avoided increasing the population's size, increasing the ratio of member that survives from generation to another, both hurting the convergence speed, or, if you don't want to slow down the convergence and keep the size of the population small, by play with the extremes-DNA-cutter to make cars explore better solution.

Also the statistical growth of the hypotheses thanks to the inertia has reduced the time spent in learning almost constant part of the track, such as unidirectional curves and straight roads.

Convergence

The described algorithm allows the cars to find a very good solution also for very particular tracks.
They are able to converge also in the presence of obstacles like the following:

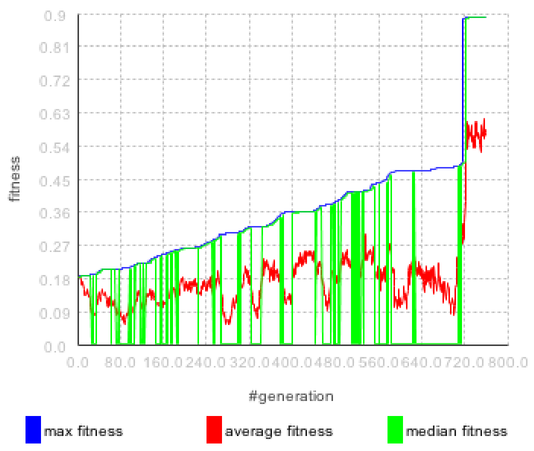


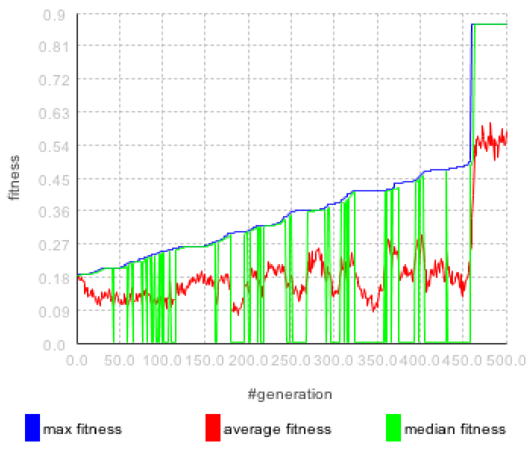


The convergence speed is almost always high, allowing finding a good solution after 200-1000 generations, depending on the difficulty of the track and on the growth constant.
In general, cars tend to learn a number of actions at each generation equal to the average of the expected values of the growth constant.

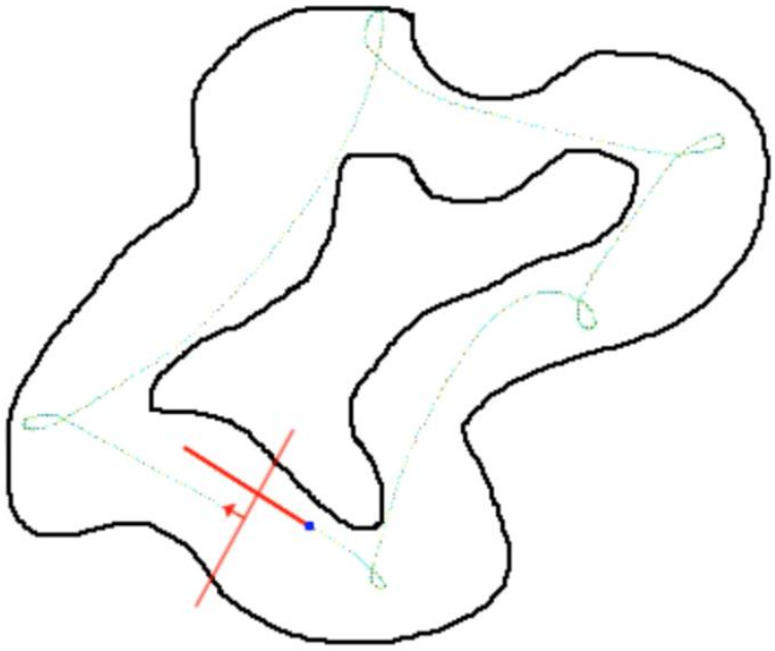
The following two tests highlight how the growth factor can speed up the convergence hurting the optimality of the solution found:

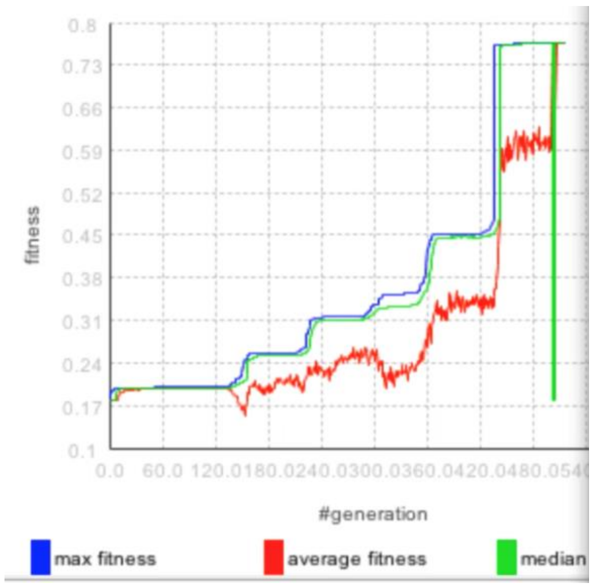
	Small growth factor	**Medium growth factor**
Genetic operators:	• cut/recreate 5 ± x, with x ∈[0, 5]; • cut/recreate 10 ± x, with x ∈[0, 5];	• cut/recreate 3 ± x, with x ∈[0, 8]; • cut/recreate 5 ± x, with x ∈[0, 10];
End reached at:	Generation #718	Generation #460
Fitness when reached:	0.8872	0.8627
#moves needed:	1951	2304
#moves/gen	2.7	5
Fitness at gen. #718	0.8872	0.8639 (Local minimum)





Another important test consisted in making the cars growing without the possibility to turn right in a clockwise track.
As shown by the below image, the result was very interesting:





Cars start turning 360°-α° to left at every curve in order to turn α° to right! It is a very important result because it shows that the algorithm is able to overcame almost significant local minima: in fact cars have to stray from the internal edge, bringing down their fitness, that only after 50-60 generations will restart increasing!

Framework

Modularity

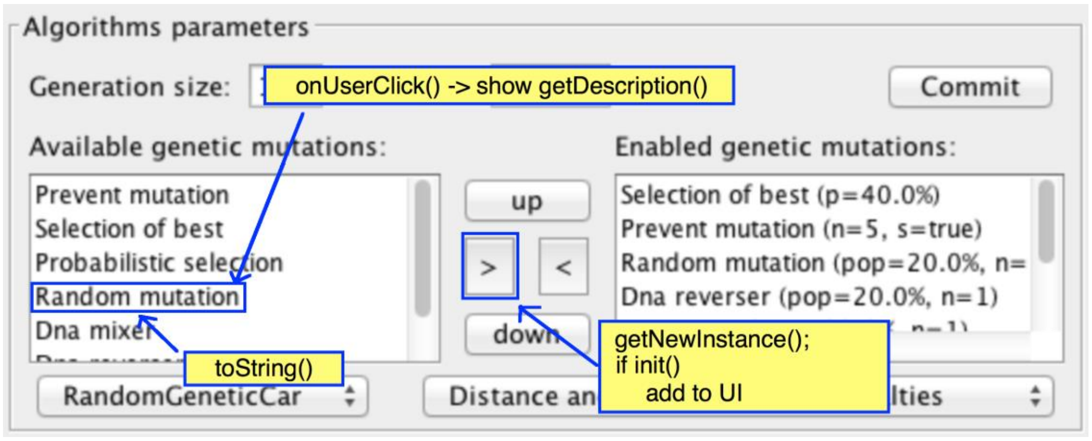
The development of this project led to the implementation of a little framework for genetic algorithm on a simple 2D environment.
The various design patterns implemented have made the code very modular, making simple to extend or modify it.
Change a track means just drawing it with a tool like Paint, but also more complex operations, like adding a new genetic operator, are simple:

```java
DrivingSchool drivingSchool = new DrivingSchool();
List<GeneticMutation> mutations =
drivingSchool.getGeneticMutationsManager().getAvailableGeneticMutation();
mutations.add(new GeneticMutation() {

    @Override
    protected List<GeneticCar> performMutationOn(List<GeneticCar> oldGeneration){
        ...
    }

    @Override
    public boolean init() {
        ...
    }

    @Override
    public GeneticMutation getNewInstance() {
        ...
    }

    @Override
    public String getDescription() {
        ...
    }
});

drivingSchool.start();
```

In the above example a new genetic operator has been added to the interface, making the user able to choose it at runtime.
The interface `GeneticMutation` is composed by the following 4 methods:

- The method `performMutationOn` is the one called when the mutation have to be performed (creating a new generation);
- The method `init` is the one called by the UI and it could be used to interact with the user and initialize some parameters of the object;
- The method `getDescription` can return a String that will be shown to the user in case he is asking for some additional information about this genetic operator;
- The method `getNewInstance` have to return a new instance of itself.

Finally, the method `drivingSchool.start()` will start the framework.

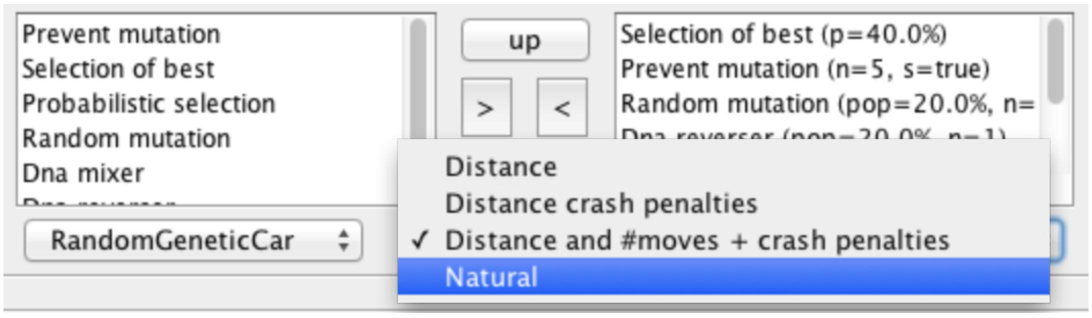

Algorithms parameters
Generation size:
onUserClick() -> show getDescription()
Commit
Available genetic mutations:
Enabled genetic mutations:
Prevent mutation
Selection of best
Probabilistic selection
Random mutation
Dna mixer
toString()
up
>
<
down
Selection of best (p=40.0%)
Prevent mutation (n=5, s=true)
Random mutation (pop=20.0%, n=
Dna reverser (pop=20.0%, n=1)
getNewInstance();
if init()
add to UI
RandomGeneticCar
Distance an
lties

If, indeed, we would like to test a new fitness function, with just few lines we are able to add it to the UI and then chose it during runtime, also while cars are running:

```
DrivingSchool drivingSchool = new DrivingSchool();
drivingSchool.getFitnessFactories().add(new FitnessFactory() {

    public FitnessFunction newInstace() {
        new FitnessFunction() {

            public boolean dependsOnOther() {
                ...
            }

            public double getFitness(GeneticCar c) {
                ...
            }
        };
    }

    public void init() {
    }
});
drivingSchool.start();
```

That's it! We have created a new FitnessFactory that return a FitnessFunction, that must provide the following two methods:

- `dependsOnOther`: return `true` if does not depend on other cars, `false` otherwise. It is useful to let the framework know that this fitness function is parallelizable.
- `getFitness`: given a GeneticCar must return a double that represents its fitness.

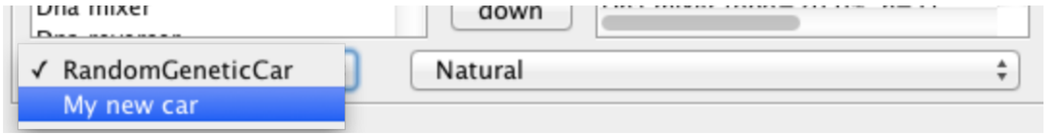


We can also add a completely customized new car, with its own actions, DNA representation,

```
DrivingSchool drivingSchool = new DrivingSchool();
drivingSchool.getGeneticCarFactories().add(new GeneticCarFactory() {

    public GeneticCar getNewGeneticCar(Engine2D engine2D, Point initialPosition,
double initialTheta, int maxNumberOfTurns) {
        return new SimpleGeneticCar(maxNumberOfTurns, engine2D, initialPosition,
initialTheta, maxNumberOfTurns) {

            @Override
            protected Action performAction() {
                ...
            }

            public GeneticCar generateNewIndividual() {
                ...
            }

            @Override
            protected double getNaturalFitness() {
                ...
            }
        };
    }

    @Override
    public double init(DrivingSchool drivingSchool) {
        ...
    }

});
drivingSchool.start();
```

ecc...
With just three methods we have created our customized car that will choose an action according to our algorithm.
The car we created extends an abstract class that provides most of the methods, letting us implementing just three of them:

- `performAction`: returns the action that should be performed on the current turn;
- `generateNewIndividual`: simply returns new instance of this class;
- `getNaturalFitness`: returns a double representing its fitness, it is called only if from the UI we will chose 'Natural' fitness function.



For example the method `performAction` of the RandomGeneticCar used for the test is the following:

```
    @Override
    protected Action performAction() {
        Action action = null;
        if (this.life < this.dna.length() && this.getNumberOfCrashes() <= 3) {
            int i = Character.getNumericValue(this.dna.charAt(this.life));
            action = Action.values()[i];
            this.life++;
        }
        return action;
    }
```