Nature Inspired Computing for Gaming: Hill Climb Racing

1st Lev Permiakov
BS 2023 DS-02
Innopolis University
Innopolis, Russia
l.permiakov@innopolis.university

2nd Aleliya Turushkina
BS 2023 DS-02
Innopolis University
Innopolis, Russia
a.turushkina@innopolis.university

3rd Arina Petuhova
BS 2023 DS-02
Innopolis University
Innopolis, Russia
a.petuhova@innopolis.university

***Abstract*—This project explores the application of Nature-Inspired Computing (NIC) techniques to enhance the gameplay mechanics and AI behavior in the popular mobile game "Hill Climb Racing". By using Genetic Algorithm, Particle Swarm Optimization, and Gray Wolf Inspired Algorithm, we aimed to improve vehicle control, terrain adaptability, and opponent AI. Our findings demonstrate that NIC can enhance gameplay by introducing adaptive learning mechanisms.**

I. Introduction

The field of Nature-Inspired Computing (NIC) has gained significant attention in recent years due to its ability to solve complex optimization problems by mimicking biological processes. In addition to optimization problems, NIC is often used in different computer games to improve AI behavior, procedural content generation, and adaptive difficulty systems [1].

This project focuses on applying NIC to "Hill Climb Racing" [2], a physics-based driving game in which players navigate challenging terrains while maintaining balance and speed. The game presents an excellent case study for NIC due to its dynamic environment which requires real-time decision-making and adaptability.

The project was inspired by a video on YouTube [3] where the author created "Hill Climb Racing" implementation using JavaScript's Box2D engine. The game had options for manual control as well as the automated control via neural networks. Neural netwoks were generated and adjusted using NEAT (NeuroEvolution of Augmenting Topologies) algorithm. The improvement and development of the machine results were interesting to observe, and we decided to make a similar project with some enhancements.

In our project, we decided to test the behavior of other nature inspired algorithms to evolve neural networks used for cars control. Moreover, we wanted to create a more dynamic and interesting gameplay so that the audience could watch the AI playthrough with the same interest as the real car racing competitions.

The goal of the project was to explore different NIC algorithms (Genetic Algorithm, Particle Swarm Optimization, Gray Wolf Inspired Algorithm) to improve the mechanics of the game and the behavior of artificial intelligence in it. The project included optimizing physics, generating new maps, and improving artificial intelligence strategies.

II. Related Work

A. Creating a game environment for the project

The first thing we started our work with was this video [3], which inspired us to create this project. We studied the author's experience in creating a simulation of the game "Hill Climb Racing". The original plan was to find the game's code in the public domain to make our simulation as close as possible to the game itself. It wasn't publicly available, so we researched how difficult it would be to write the game ourselves. The first options were Unity and Python. When we looked at the option of creating a Python game, we ran into the problem that there were no libraries that could simulate the physics of the game world, and this is a very important detail that we would not be able to do ourselves. The second option was Unity, but since we had no experience in writing games with this engine. We decided to study the moment when the game is written in the video. In the video we found the author's open repository with implemented game mechanics [5]. Then we found other options on github, for example [4]. After looking at the implementation in that repository, we decided to take the implementation of the game from [5] and adapt it for us. Having decided on the [5] version, we decided to study how much the game differed from the original, and first made a test version with manual controls. To understand how the physics work, how the maps are built, which parameters can be used as input to a neural network, or you can set a combination of commands for movement. After studying the maps, we came to the conclusion that they would not be able to go through any combination of movements because the terrain is too varied.

B. Changes in the borrowed code

Since we were testing how the machine would be controlled at the beginning, we also changed the map generation slightly. Because our first cars could not go anywhere, in principle, because of a simpler structure than the author's. However, simplifying the map did not make it easier to complete. We also slightly changed the speed to make it more similar to the original.

C. Neural Networks

In the video [3], NEAT is a genetic algorithm that creates and develops artificial neural networks, evolving their topology and weights. The main features are that the neural network divides the population into subgroups, each of which solves its own task, which helps to maintain innovation, and gradual complication, which consists of adding new neurons and connections as needed. In the video, [3] uses NEAT to teach an AI to drive a car in the game Hill Climb Racing. It creates a population of neural networks, each of which controls the machine and makes decisions based on the current state (e.g. position, speed, tilt angle). The networks are rated according to their ability to negotiate the course, and the best ones mate and mutate, creating a new generation. So we can conclude that the creator was inspired by the genetic algorithm. We took a simpler neural network, the feed-forward neural network. This neural network continues

D. Natural algorithms for determining weights in a neural network

At first we just wanted to change the weights in the neural network using the genetic algorithm, as we had seen examples of its implementation. When we first started the machines, they didn't show any significant results. So we had to pay a lot of attention to the parameters and their selection. As a result, with the help of a genetic algorithm, we were able to get the cars to complete the route in 30-40 epochs. Having found this indicator, we decided to look at other algorithms, analysing them and paying particular attention to how well they were suited to fast and complex tasks. We chose Gray Wolf Optimizer (GWO) and Particle Swarm Optimization (PSO), but they didn't get very far without a long selection of parameters. As a result, we were able to use PSO to complete the map in 7 rounds, and we noticed a tendency in his work that no matter what parameters we set, he either passed the game in the first 20 iterations or did not pass in principle. This is due to the fact that he tends to find local maxima and it is difficult for him to get out of them. GWO, like the Genetic Algorithm, is able to complete the map in 30-40 rounds by applying a strong mutation to this algorithm in proportion to its place in the list. Strong mutation helps the algorithm to get out of local maxima and allows it to take more cars to the end of the route.

III. Methodology

Our methodology includes the implementation of three different nature-inspired computational algorithms: Genetic Algorithm (GA), Gray Wolf Optimizer (GWO) and Particle Swarm Optimization (PSO). We used several different algorithms to examine the effectiveness of each algorithm for completing the Hill Climb Racing game [2]. Each algorithm offers a unique optimization strategy based on natural phenomena.

A. Neural Network

The neural network implemented in the NeuralNetwork class has an input layer of 5 neurons (defined when the network is created), a hidden layer of 4 neurons (hiddenNodes), and an output layer of 2 neurons (forward or backward movement).

When creating the network, the weights are initialized with small random values (-0.3 to 0.3) using the randomMatrix() method, and the biases are initialized with a value of 0.1.
The main work of the network takes place in the predict() method. In the hidden layer, each neuron receives a weighted sum of inputs, and an offset is added. In the output layer, the neuron receives a weighted sum of the hidden layer outputs, adds an offset, and returns a linear output.
A neural network is used in the Player class to control a car. First, using the getCarState() method, the car receives input data: the car's X position, X and Y speed, the car's tilt angle, and the wheels touching the ground flag. Second, using the AIControl() method, the machine makes a decision: the output of the network is scaled (×0.5). If the output ¿ 0.1, the forward motor is turned on. If the output is ¡ -0.1 , the motor backward is turned on. Otherwise, the motor is turned off.

Thus, this neural network is a classic example of a simple but effective implementation for control tasks in simulations where learning is performed using nature-inspired algorithms.

B. Genetic Algorithm

This project implements a genetic algorithm for training neural networks that control cars in the Hill Climb Racing game [2].

The genetic algorithm initializes the initial population generation and creates a random landscape. Then it begins an evaluation phase where it tests all individuals, records the maximum distance traveled, and determines fitness values. Next, individuals are ranked, genetic operators are applied, and a new population is generated.
In our genetic algorithm, each individual of the population is represented by an instance of the class Player, an associated neural network (brain), and phenotype parameters (color characteristics). The fitness function is determined by the formula fitness = $\max\left(1, \left\lfloor \frac{d_{max}-349}{10} \right\rfloor\right)$, where d_{max} is the maximum distance traveled and $constant$ 349 represents the initial offset.
We use two genetic operators: mutation (mutate method in NN.js) and crossover (crossover function in sketch.js).
Mutation is applied to the weights and biases of the neural network with a probability that depends on the rank of the individual: $p_{mut}(i) = 0.1 + 0.2 \cdot \frac{i - eliteCount}{N - eliteCount}$, where i is the index of the individual after sorting, $N = 50$ is the population size.
We select the top 20 individuals for crossover, which creates a new network by combining the weights of two parents with a probability of 0.5:

$$w_{ij}^{child} = \begin{cases} w_{ij}^{parentA} & \text{if } r < 0.5 \\ w_{ij}^{parentB} & \text{otherwise} \end{cases}$$

For each weight, a value from one of the parents is randomly selected, and a small additional mutation is added with probability 0.1: $w_{ij}^{child} := w_{ij}^{child} + \delta, \quad \delta \sim U(-0.2, 0.2)$
The selection function ranks the entire population by fitness function, retains the 10 fittest individuals, and generates 40

new individuals. Individuals are generated by tournament selection of parents, where the tournament size is two, crossover and mutation.

Our realization has its own peculiarities. First, the parameter $steepnessLevel$ provides a gradual increase in the complexity of the problem: $s(d) = 100+80 \cdot \frac{d}{d_{max}}$. Second, storing the best $bestScores$ and visualizing the generational progress. Third, realistic dynamics using Box2D and accounting for deflection and surface contact angles.

Consequently, we implemented a genetic algorithm that allows neural networks to gradually learn how to efficiently drive a car in an increasingly complex environment.

C. Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an optimization technique inspired by the behavior of a flock of birds or a school of fish. In this project, PSO is used to train neural networks that control cars in the Hill Climb Racing game [2].

In the implemented algorithm, the particle is represented by a Player class with a neural network, and the fitness function is the distance traveled by the car (player.score).

The particles are initialized as follows: weights are initialized with random values in the range [-0.1, 0.1] using the randomMatrix() method, velocity matrices (velocity-ih, velocity-ho) with zero values are created, and the best weights (bestWeights-ih, bestWeights-ho) are remembered. The updateParticle() method is used to update the particles. Each particle remembers its best state (bestWeights-ih, bestWeights-ho).

In the main loop of the algorithm (sketch.js):

1) New generations are selected and created: the elite individuals (first 9) are transferred to the new generation unchanged, and the rest of the particles are created based on the previous generation using PSO.

2) The particle with the maximum $bestScore$ is found. Its weights are used as a global best solution to update other particles.

3) Each generation is evaluated by the distance traveled; after all cars are killed, a new generation is created.

The PSO implementation in this project solves the problem of training neural networks for car control. The algorithm demonstrates a balance between exploring the solution space and exploiting the beneficial solutions found, allowing cars to gradually improve their performance in the challenging environment of Hill Climb Racing. However, this algorithm is not efficient, as it is more likely to hit a local minimum and not progress further.

D. Gray Wolf Inspired Algorithm

Another algorithm for training a neural network that we used in the comparison is the Gray Wolf Inspired algorithm.

At the beginning, an initial population of 50 players is created with random neural networks, and random terrain is generated (Ground.randomizeGround()). Each player controls a machine using its own neural network (AIControl()). The neural network receives input about the state of the car (getCarState()) and decides to move. Players get points for the distance traveled (score). When all players have died (allDead), nextGeneration() is called. The top 8 players (leaders α, β and δ) move to the next generation unchanged. The remaining players are created based on the top eight, but with mutations.

Mutation in the algorithm occurs so that the weights of the neural network are randomly changed (mutateBrain()), and the frequency and strength of mutations depend on the rank of the player in the population. The process is repeated for each new generation, improving the average distance traveled.

The Gray Wolf Inspired Algorithm gradually improves the cars' control by selecting the best neural networks and combining them with mutations. This allows the cars to drive farther and farther with each generation.

IV. GitHub Link

We have publicly posted the project code on GitHub:

https://github.com/arinapetukhova/HillClimbRacing

This repository contains the full implementation of the algorithms, configuration files, and documentation necessary to reproduce our experiments.

V. Experiments and Evaluation

Most of the experiments were done due to the necessity to tune the parameters of the algorithms. For all the algorithms, we decided to keep the population size of 50 members to make the program run quite fast and at the same time to ensure the diversity among the population.

A. Genetic Algorithm

After trying different values for other parameters, we decided to use the best 20 cars as parents in the crossover stage while preserving only the best 10 of them in the next generation. This approach was good in terms of exploiting the best old solutions and not converging quickly to a mediocre results at the same time.

As mutation rates, we used adjustable values (scaled according to the rank of the car in the population in comparison to the best solutions) from 0.1 to 0.3, which helped to control exploration/exploitation intensities depending on each car's performance. Mutation strength was chosen to be 0.3 to avoid too large changes and provide new, possibly better, solutions.

Testing showed that on the first generations the score was improving consistently but not so quickly. On further generations, the score improvement speed was slowing down but the consistency was preserved. At some point, one car was reaching the end. Then, some next generations either showed a little worse results or the same result of only one car reaching the end. After several generations, more cars were beginning to reach the end. In the best test, a car reached the end of the map on the 22nd generation.

B. Particle Swarm Optimization

After trying different values for other parameters, we decided to keep the 9 best cars in the next generation to balance research and operations. The neural networks of the other cars were updated according to one of the best global solutions. Parameter 1.3 was chosen as the inertia weighting coefficient so that the particles would explore the space next to the leader more aggressively and not get too close to it. In order for the particles to retain their uniqueness and not become completely similar to the leader, the parameter 2.5 was chosen so that the particles would trust their experience. However, in order to keep them moving in the right direction, a social efficiency of 1.2 was chosen. These parameters proved to be the best, as the particles explored the area near the leader and gradually improved.

In one of the tests, the car reached the end of the map using the 7th generation algorithm. However, other tests have shown that in most cases the results in subsequent generations tend to reach local minima without further improvement if good solutions were not obtained in the first generations.

C. Gray Wolf Inspired Algorithm

After trying different values for other parameters, we decided to preserve the top 8 best-performing cars in the next generation. The other cars' neural networks were updated depending on the randomly chosen solution from these 8 best cars. After copying the neural network of the chosen car, mutation was applied to the offspring. This approach maintained high-performing solutions while still allowing sufficient diversity through mutation of offspring. The elite count of 8 was chosen as a balance between preserving good solutions and preventing premature convergence.

For mutation parameters, we implemented a rank-based adaptive system where mutation rates scale from 0.1 to 0.7 depending on a car's position in the population ranking. Lower-ranked cars receive higher mutation rates to encourage exploration, while offspring of elites receive minimal mutation. We fixed the mutation strength at 0.4, which proved optimal for making meaningful weight adjustments to the neural networks without causing destabilizing changes to the driving behavior.

During testing, we observed that on early generations the score was improving faster than in GA as the population explored different driving strategies. On further generations, the score improvement speed was slower than in GA, but consistency was preserved. At some point, one car was reaching the end. After that, next generations either showed the same results or more cars were beginning to reach the end. In our best test run, the first car reached the end by generation 25.

VI. Analysis and Observations

The cars showed different behavior through the game process depending on the algorithm used. After testing the chosen algorithms, we got the following results.

A. Genetic Algorithm

The Genetic Algorithm used selection, where the top 20 best-performing cars (according to the scores achieved) from each generation were used for crossover to produce offspring for the next generation. This approach effectively eliminated poorly performing members while preserving and refining successful strategies. Over successive generations, the population demonstrated improved adaptability, with more cars successfully navigating the terrain.

The mutation rate was adjustable and ensured exploitation of existing high-performing traits for cars with higher score, while occasional mutations introduced exploration by testing new control strategies for cars with lower score. The adjustable mutation helped to prevent instability and premature convergence related to mutations with some fixed rate.

The early generations favored exploration by testing diverse strategies. Later generations shifted toward exploitation by fine-tuning the best strategies. That is why, initially, only one car reached the end of the track on 30th - 40th generations. After another 5 - 10 generations, knowledge spread through crossover, leading to more cars successfully completing the map. However, it is possible that after the first car has reached the end, the next generations might have less successful results before more than one car starts finishing the map. It happens due to the randomness of the crossover function.

B. Particle Swarm Optimization

The Particle Swarm Optimization algorithm was the worst performing of the approaches tested due to fundamental limitations in its design. Unlike genetic algorithms, which use crossing and mutation, PSO relies entirely on tracking the single global best solution (gbest), a group of leaders that pass from generation to generation, and the memory of individual particles (best). This approach led to shortcomings in convergence and diversity. The performance of the algorithm depended largely on the random initialization of the first generations. As there are no genetic operators in PSO, such as crossover and mutation, the whole population simply tended towards any decision that happened to emerge as the first leader. This often led to the premature adoption of sub-optimal strategies when the initial result was mediocre. Therefore, if no successful cars were produced initially, the results soon became stuck at some local lows and stopped improving steadily in subsequent generations. However, because the algorithm had fast convergence, it could produce a fast solution in 7-10 moves. If a solution has not been found after the 25th move, it is very likely that it will not be found because the cars are starting to travel a smaller distance.

C. Gray Wolf Inspired Algorithm

The Gray Wolf Inspired algorithm used a hierarchical selection mechanism, where the top 8 best-performing cars (α, β, and δ leaders) guided the evolution of the population. This structure preserved elite strategies while systematically eliminating poor performers. Over generations, the population showed accelerated adaptability, with cars increasingly

mastering complex terrains through the combined influence of leader-based guidance and rank-driven exploration.

The mutation rate was dynamically scaled based on performance rank, ensuring strong exploitation of high-performing traits (low mutation for elites) while promoting exploration in weaker candidates (high mutation up to 70 percents). This adaptive approach prevented stagnation and maintained diversity without destabilizing convergence. The early generations emphasized exploration through high mutation in lower-ranked cars, while later generations focused on refining elite strategies through leader-directed weight blending and minimal mutation.

Initially, only the α-wolf (top performer) could complete the track on the 35th - 45th generations, but after another 1 - 5 generations, the effect of leader guidance and adaptive mutation enabled more cars to successfully navigate the map. Therefore, in later generations, the "pack intelligence" allowed lower-ranked solutions to inherit and refine successful traits from the $\alpha/\beta/\delta$ hierarchy faster than the crossover related adjustments in GA.

VII. Conclusion

Our project has produced many different solutions, each of which fulfils a different task: Particle Swarm Optimisation allows you to find a solution quickly, but it will not always be able to do it, Genetic Algorithm allows you to do a comprehensive search for solutions that can provide an answer at an average speed, Gray Wolf Inspired allows you to get many solutions over a long time that are somewhat similar, but each one is unique. These algorithms help to optimise the weights of the neural network and make good controls for enjoying the evolution of cars in the game "Hill Climb Racing". Future plans for our project may include upgrading the neural network and then writing more complex algorithms.

References

[1] R. Marks, "Playing games with genetic algorithms," in Proc. IEEE Conf. on Evolutionary Computation, Accessed on: Apr. 23, 2024, 2002. [Online]. Available: https://www.researchgate.net/publication/228697986_Playing_Games_with_Genetic_Algorithms.

[2] Fingersoft, Hill climb racing, Accessed on: Apr. 23, 2024, 2011. [Online]. Available: https://play.google.com/store/apps/details?id=com.fingersoft.hillclimb&hl=ru.

[3] Unknown, A.i. learns to play hill climb racing, Accessed on: Apr. 23, 2024, 2018. [Online]. Available: https://youtu.be/SO7FFteErWs?si=NIiU9K-63v_0FHNz.

[4] M. Amin, Hill–racer, Accessed on: Apr. 23, 2024, 2015. [Online]. Available: https://github.com/Manini00/Hill–Racer.

[5] C. Bullet, Hill-climb-racing-ai, Accessed on: Apr. 23, 2024, 2024. [Onlne]. Available: https://github.com/Code-Bullet/Hill-Climb-Racing-AI.