Autonomously playing Yellow-Spaceship using NEAT and GP

Samuel Bortolin, 221245, *samuel.bortolin@studenti.unitn.it*
Davide Lusuardi, 223821, *davide.lusuardi@studenti.unitn.it*

***Abstract*—Game playing was an area of research in AI from its inception. Video games offer an amazingly interesting testbed for AI research and new ideas. The aim of this report is to show the potential of NeuroEvolution of Augmenting Topologies and Genetic Programming in solving the Yellow-Spaceship game and compare the two strategies. We found out that both are very good approaches and managed to find a way to play the game well. Even though, across multiple runs, NEAT seems to obtain more stable results, GP has proved to have a bigger potential thanks to the tree-structured individuals and overall manages to reach very good results.**

***Index Terms*—NEAT, Neural Networks, Neuroevolution, GP, Genetic Algorithm, Artificial Intelligence, Computer Games, Autonomous agent.**

I. Introduction

COMPUTER GAMES have been linked with artificial intelligence (AI) from its inception. Video games implement rich and complex environments that usually require a real-time response by the player and a dynamic strategy with an adaptive behavior in order to get a good score or win.

Artificial Neural Networks (ANNs) are computational learning systems inspired by biological neural networks that use a network of functions to understand and translate some input data into a desired output. An ANN is based on a collection of connected units or nodes called artificial neurons, each neuron may be in principle connected to all the others. We can differentiate the neurons in three categories: input neurons, which receives the input data from the environment; hidden neurons, which elaborates the input data; output neurons, which outputs the desired output.

They are being deployed in a variety of tasks including playing computer games, but a core problem with the standard training approach is that we do not know a priori the best structure and we may have to try a lot of different architectures before finding the one that performs well. NeuroEvolution of Augmenting Topologies (NEAT) [2] is a genetic algorithm, an example of a topology and weight evolving artificial neural network (TWEANN), which attempts to simultaneously learn weight values and an appropriate topology for a neural network. In our implementation [7], NEAT is used to evolve an ANN that is evaluated during the game to output the action the agent has to perform.

Genetic Programming (GP) [3] is a technique used to generate computer programs. Starting from an initial population of programs, the GP algorithm will evolve them in order to solve predescribed tasks and machine learning problems. In our implementation [7], GP evolves computer programs represented as tree structures, that are evaluated recursively during the game to output the action the agent has to perform.

II. Scenario and Problem

Yellow-Spaceship [1] is a space shooter game implemented using Pygame [6] in which the player, a battleship, fights a number of enemies by shooting at them while dodging their fire: enemies can be aliens or spaceships. The player and the enemies move horizontally and can fire: actions can be defined as move left, move right, do not move, fire; moving and firing can be performed simultaneously, so in the end, we have in total 6 possible combinations of actions. Levels are structured such that there are always 2 aliens in each level except for levels multiple of 5 where, instead, there is an enemy spaceship. The battleship and the enemy spaceship have an initial health of 50 life points whereas the health of enemy aliens increases as the level increases. Initially, the alien health is set to 10 life points and gradually increases by 8 life points every 10 levels. The laser damage for the battleship is fixed to 1 life point whereas the one of enemies is initially set to 2 life points and gradually increases by 6 life points every 10 levels. As the game proceeds, it becomes more difficult to beat the enemies, enemy lasers become more lethal and each level takes longer to be completed. Fig. 1 shows a screenshot of the game.

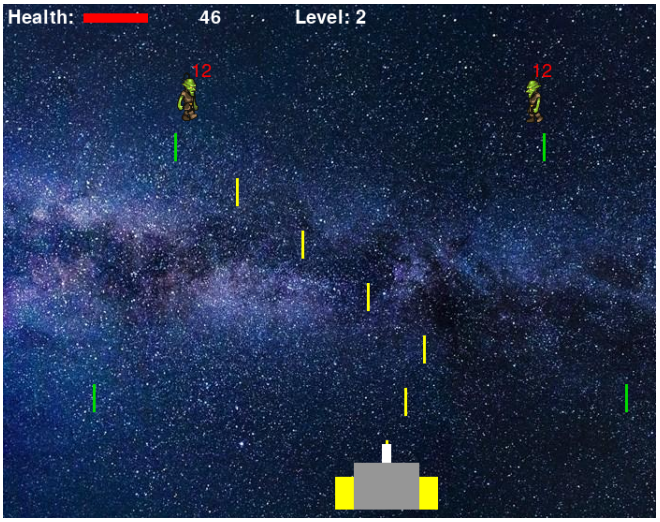


Fig. 1: Screenshot of the Yellow-Spaceship game [1].

The number of levels is not bound and the aim of the game is to kill as many enemies as possible and reach the highest level without losing the whole life.

III. Methodologies and Implementation

Starting from the Yellow-Spaceship game [1], we have adapted the code to our needs. In particular, we have modified the code that reads the action from the user keyboard in order to be able to substitute the user input with an action computed by a program and apply it to the game. The controlling action can be computed by an ANN evolved using NEAT, or by a tree-based program evolved using GP, passing them the appropriate information about the current state of the game. Moreover, the possibility to not show the game has been introduced in order to speed up the execution.

The inputs from the game environment for both NEAT and GP individuals are the following: the x coordinate of the battleship; the velocity of the battleship; the x coordinate of the first and second alien (if any, otherwise they are set to 0); the x and y coordinates of the first and second closest lasers (if any, otherwise they are set to 0); the x coordinate of the enemy spaceship (if any, otherwise it is set to 0). In this way, there are a total of 9 arguments of type float passed to the individuals as input that can be used to decide the action to perform. The performance of an individual can be evaluated based on how much it moves on in the game.

The fitness function for the individuals of both the algorithms that we decided to use can be formulated as follows:

$$\begin{aligned} fitness = & alien\_kills * 10 + \\ & enemy\_spaceship\_kills * 50 + \\ & \sum_{h \in battleship\_healths} \frac{h}{50} + \\ & \sum_{a \in aliens} \frac{cur\_alien\_health - a.health}{cur\_alien\_health} * 10 + \\ & \sum_{s \in enemy\_spaceships} 50 - s.health \end{aligned} \tag{1}$$

where $alien\_kills$ is the number of aliens killed, $enemy\_spaceship\_kills$ is the number of enemy spaceships killed, $battleship\_healths$ is a vector of battleship health at the beginning of each level and cur_alien_health is the initial alien health at the last level reached.

The fitness function should be maximized and takes into account the number of aliens and enemy spaceships killed, the battleship health at each level and the health of aliens and enemy spaceships at the last level reached. There is no intermediate reward and the overall fitness is returned at the end of simulation.

In order to handle cases in which the simulation takes too much time because the battleship is not killed, we introduced a maximum threshold on the number of frames: the game will be stopped after 1 million frames. The threshold permits to prevent the execution from getting stuck and also incentivizes the NEAT and GP algorithms to learn how to reach higher levels with the same number of frames. When the best individual execution is shown, this limit is released in order to obtain the actual fitness without interrupting the game.

To speed up the evolution process, individuals are evaluated only every 10 frames to obtain the action to perform: the previous action is applied for the rest of 9 frames. This value should be sufficient to permit the individual to have enough control over the battleship actions and not be killed by enemies. In this way the evaluation process proceeds about 10 times faster and the battleship movements look smoother.

A. NEAT

The implementation is based on the NEAT-Python library [4], which is a pure Python implementation of NEAT, with no dependencies other than the Python standard library. We decided to evolve a feed forward neural network with the possibility of learning skip connections instead of a recurrent neural network since the actual state of the game is sufficient to evolve the neural network and no more information is required. The network has 6 output nodes and each of them encodes a particular action that the agent has to perform: go left; go left and fire; stay still; stay still and fire; go right; go right and fire. The output node with the maximum value is taken to decide the action to perform.

We decided to adopt the following algorithm parameters:

- $num\_runs = 10$, $num\_generations = 200$ and $no\_fitness\_termination = True$
- $pop\_size = 100$ and $reset\_on\_extinction = True$
- $activation = sigmoid$ and $aggregation = sum$
- $conn\_add\_prob = 0.5$ and $conn\_delete\_prob = 0.5$
- $enabled\_default = True$ and $enabled\_mutate\_rate = 0.01$
- $feed\_forward = True$ and $initial\_connection = full\_nodirect$
- $node\_add\_prob = 0.2$ and $node\_delete\_prob = 0.2$
- $num\_inputs = 9$, $num\_hidden = 9$ and $num\_outputs = 6$
- $species\_fitness\_func = max$, $max\_stagnation = 20$ and $species\_elitism = 2$
- $elitism = 2$, $survival\_threshold = 0.2$ and $min\_species\_size = 2$

All the configuration parameters used by the algorithm can be modified in the configuration file named 'configNEAT.txt' present in the root directory of the project.

B. GP

The implementation is based on the DEAP library [5], which is a Python evolutionary computation framework which provides the main blocks for building Genetic Programming algorithms. Strongly typed genetic programming (STGP) is an enhanced version of genetic programming which enforces data type constraints.

A tree-based program has been evolved using STGP. The output of the program is one of the 6 encoded actions that the agent can perform. To encode the actions, we defined one class for each of them with the following encoding: A, go left; B, go left and fire; C, stay still; D, stay still and fire; E, go right; F, go right and fire.

The terminal set is composed of the action classes, some float values (5, 10, 15, 20, 25, 30) and the boolean values ("true", "false"). The function set is composed of the boolean

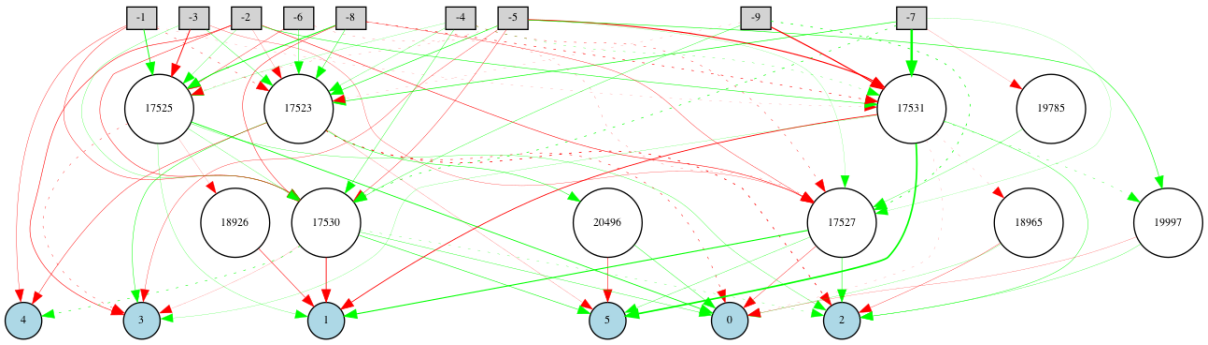


Fig. 2: Best ANN generated by NEAT.

operators "greater than", "equal", "and", "or" and "negation", the arithmetic operators "add", "subtract" and "multiply" and the "if_then_else" function that can outputs a float value or an action class. Since the "if-then-else" function is the only function that outputs an action, it should always be at the root of the tree.

We decided to adopt the following algorithm parameters:

- $num\_runs = 10$ and $num\_generations = 200$
- $pop\_size = 100$
- $crossover\_prob = 0.5$
- $mutation\_prob = 0.5$
- $hof\_size = 4$
- $max\_tree\_size = 500$
- $max\_tree\_height = 10$
- As $expr\_init$ we used $gp.genFull$ with $min\_ = 1$ and $max\_ = 3$
- As $select$ we used $tools.selTournament$ with $tournsize = 7$
- As $mate$ we used $gp.cxOnePoint$
- As $expr\_mut$ we used $gp.genFull$ with $min\_ = 0$ and $max\_ = 2$
- As $mutate$ we used $gp.mutUniform$
- As $algorithm$ we used $deap.algorithms.eaSimple$

All the configuration parameters used by the algorithm can be modified in the configuration file named 'configGP.txt' present in the root directory of the project.

IV. Results

In this section we will present and compare the results obtained with the two approaches. Some results obtained from a randomly piloted battleship are considered as baseline.

For both NEAT and GP, the evolution process stores at each run the best program generated with the relative statistical results and plots in the 'runs' folder. In this way, it is possible to launch the program later and observe through the game interface how the battleship behaves.

A. NEAT

The NEAT algorithm managed to evolve a ANN that can reach level 42 in the game with a fitness of 1114 after about 160 generations. The ANN is represented in Fig. 2. As we can see the network is quite simple, with a small number of nodes and connections: it has just 10 hidden nodes and 63 enabled connections (that are represented by the solid arrows).

In Fig. 3, we can observe that the best fitness increases significantly in just a few particular generations, whereas the average fitness does not grow significantly and stays under 200. With this fitness trend, we can also notice that the speciation decreases significantly after 20 generations due to the stagnation of a lot of species: after about 60 generations, just the 2 elitism species will survive as shown in Fig. 4.

A weak point of NEAT is that the generated ANN is not really interpretable and we cannot understand why the network produces certain output values.

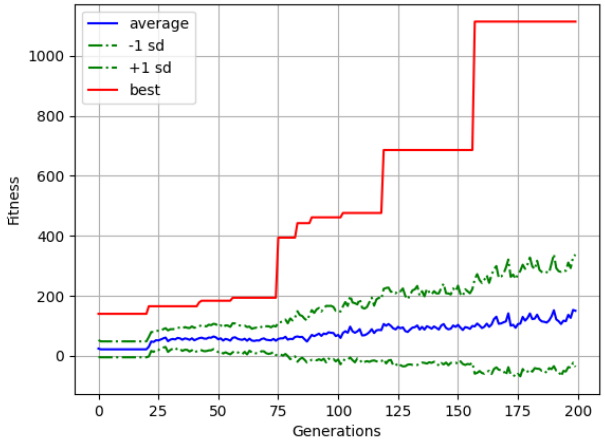


Fig. 3: Fitness trend of the NEAT run that has generated the best ANN.

B. GP

The GP algorithm managed to evolve two tree-based programs that reach over 4000 fitness in the simulation.

The first program reaches level 151 in the game with a fitness of 4061 during the evolution process and level 194 with a fitness of 5212 without limiting the number of frames. The tree of the program is represented in Fig. 5a. As we can see the tree is quite complex, with 259 nodes and a depth of 10 (the maximum depth allowed). In Fig. 6a, we can observe

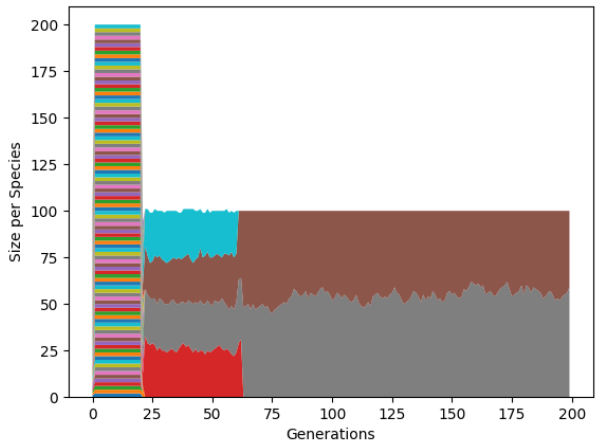


Fig. 4: Speciation trend of the NEAT run that has generated the best ANN.

that the best fitness improves a lot in the first 60 generations reaching the frame threshold.

The second program reaches level 162 in the game with a fitness of 4342 during the evolution process and level 189 with a fitness of 5076 without limiting the number of frames. The tree of the program is represented in Fig. 5b. As we can see the tree is a bit less complex than the previous one, with 177 nodes and a depth of 10 (the maximum depth allowed). As shown in Fig. 6b, the best fitness improves significantly only after 100 generations, reaching the frame threshold.

In both cases, for the remaining generations, the best fitness trend increases slightly, probably also due to this limitation on the number of frames. The average fitness trend is smoother and follows the best trend with a similar increase, reaching about 3000 fitness with a large variance (over 1000).

A weak point of GP is that the generated tree is a bit complex and can be simplified a lot, e.g. simplifying some "if_then_else" statements that have as condition pure boolean values as can be seen in Fig. 5.

C. Comparison

With both techniques, we managed to find a program that is able to play the game well, learning to fire almost always to hit enemies while dodging their lasers.

Considering our scenario, GP demonstrated to have a bigger potential: thanks to the tree-structure and the functions provided, it is able to learn more complex strategies reaching level 194, much more than the best evolved ANN. NEAT, instead, struggles to evolve the network reaching at most level 42. NEAT is probably more suitable for tasks where a program would not be the best option or even it would be impossible to build.

Moreover, a tree with a limited size can be more interpretable than a ANN and we can easily provide an explanation for what the agent is doing.

Observing the agent in action we can say that both NEAT and GP agents move quite smoother, even though neither of them resemble humans. GP agent, in the end, looks smarter and tends to stay in the middle of the playing area dodging all the lasers, whereas NEAT tends to remain in the corners.

Across multiple runs, NEAT seems to obtain more stable results with respect to GP that manages to reach very good results only few times, as shown in Fig. 7. Both the techniques find a program which performs much better than a randomly piloted battleship that on average is able to reach only level 3 with a fitness of about 45 as shown in Fig. 7c.

V. Conclusions

As demonstrated, NEAT and GP are very good approaches able to find a way to play the game well. Even though, across multiple runs, NEAT seems to obtain more stable results, GP has proved to have a bigger potential thanks to the tree-structured individuals and overall manages to reach very good results.

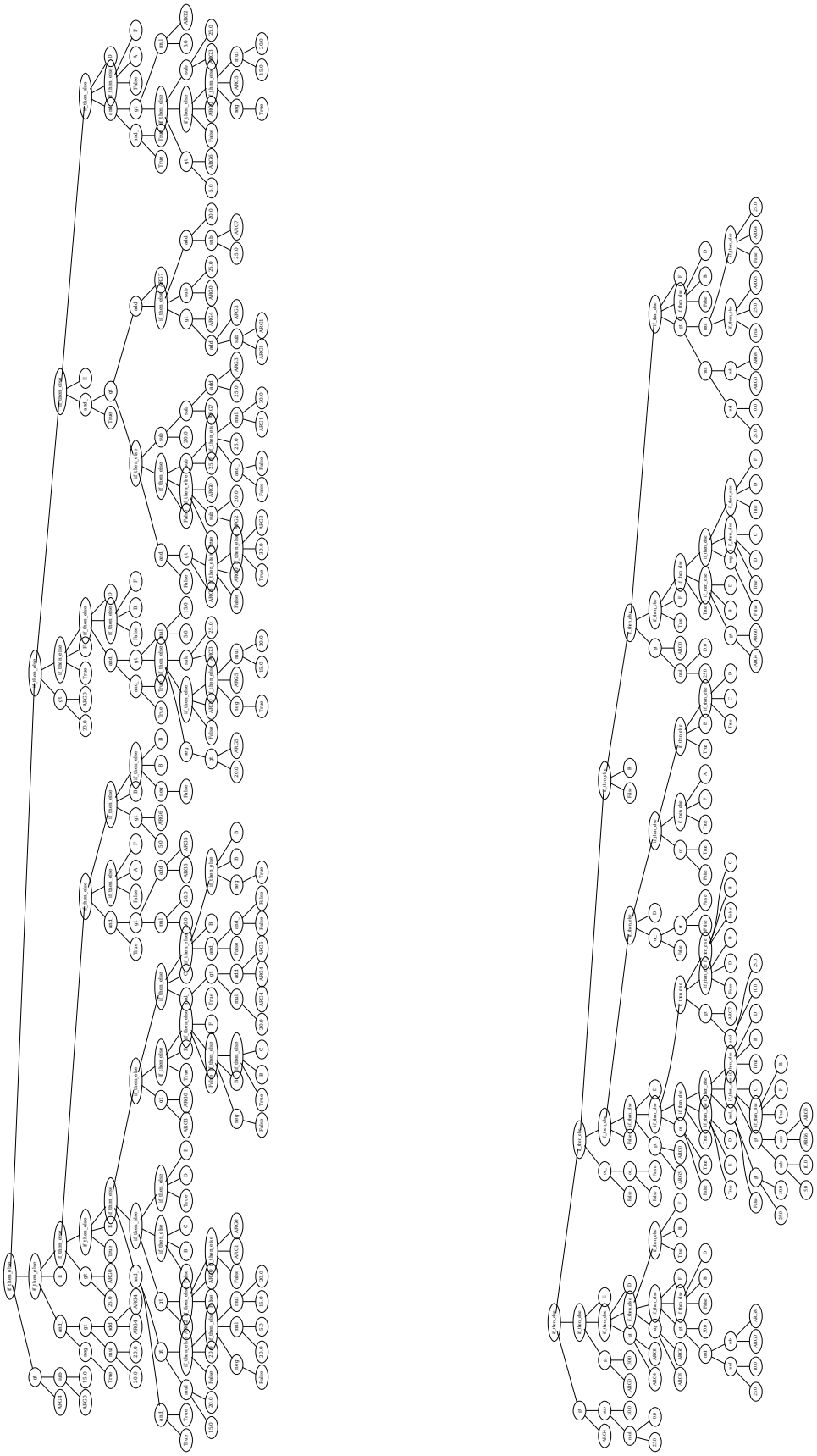
One of the issues that we encountered using GP is how to manage the different input types: one may implement functions taking into account inputs of different types or, alternatively, can use a strongly typed primitive set as we have done.

Another issue is that the execution of the algorithms takes a lot of time and computational resources, especially when the frame threshold is reached. Due to this, the number of runs and individuals should be limited.

In the end, with this project we learned a lot about this field, in particular how it is possible to apply bio-inspired techniques to real-world problems and benefit from them. They have proved to be a valid alternative to classic back-propagation algorithms for ANN and we have learned how to apply them to Reinforcement Learning tasks using the fitness as a reward.

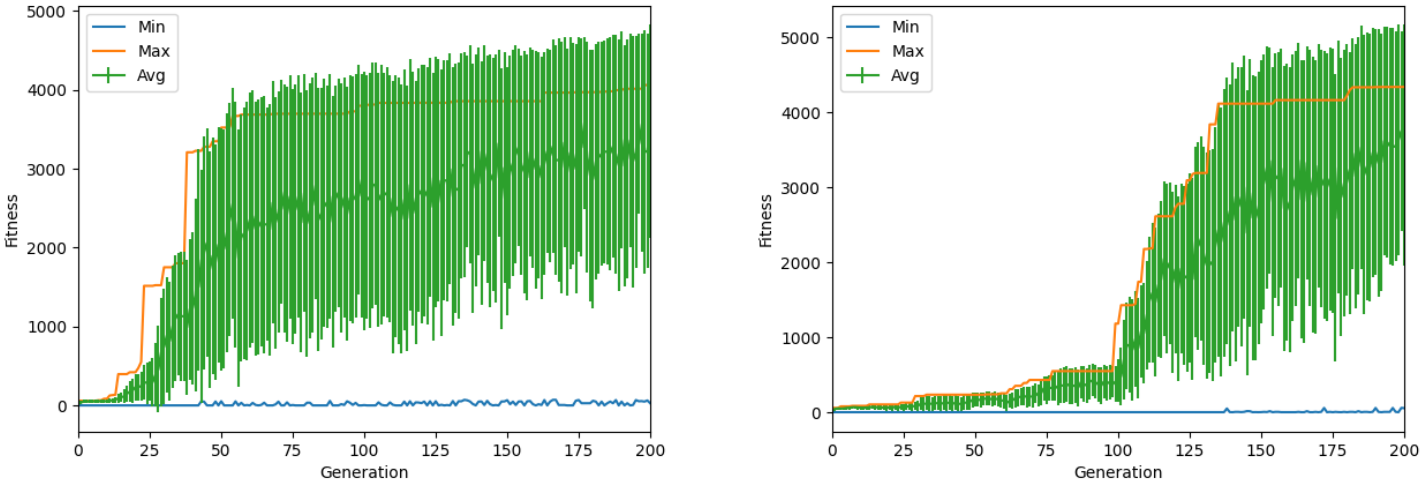
References

[1] https://github.com/ph3nix-cpu/Yellow-Spaceship
[2] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," in Evolutionary Computation, vol. 10, no. 2, pp. 99-127, June 2002, doi: 10.1162/106365602320169811.
[3] Koza, John R. "Hierarchical genetic algorithms operating on populations of computer programs." IJCAI. Vol. 89. 1989.
[4] https://neat-python.readthedocs.io/en/latest/index.html
[5] https://deap.readthedocs.io/en/master/index.html
[6] https://www.pygame.org/docs/
[7] https://github.com/samuelbortolin/Bio-Inspired-Spaceship



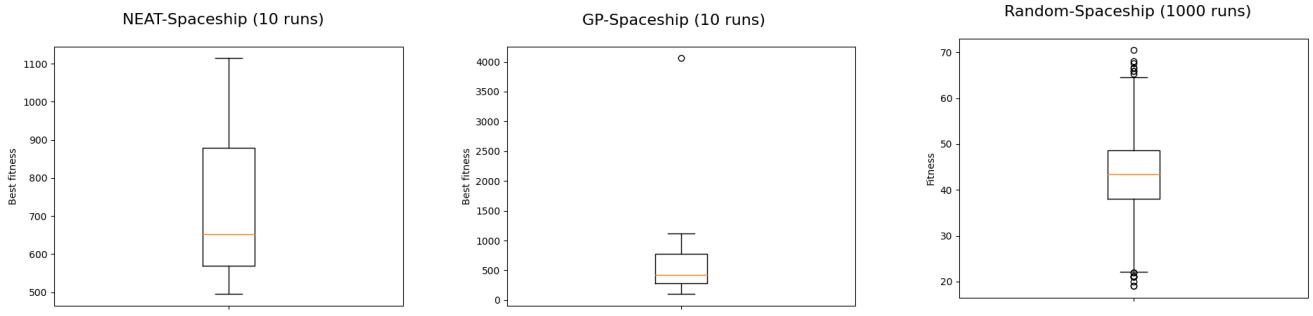
(a) (b)

Fig. 5: Best tree-based programs generated by GP.



(a) (b)

Fig. 6: Fitness trend of the GP runs that have generated the best programs.



(a) NEAT boxplot. (b) GP boxplot. (c) Random-piloted battleships boxplot.

Fig. 7: Comparison between the boxplots of NEAT, GP, and randomly piloted battleships.