Evolutionary Computing - Task I: The Specialist Agent (Group 51)

October 1, 2023

Jade Dubbeld
(11692065/jdd360)
University of Amsterdam
Amsterdam, The Netherlands
j.y.g.dubbeld@student.vu.nl

Noah Knijff
(11882336/nkf260)
University of Amsterdam
Amsterdam, The Netherlands
noah.knijff@student.uva.nl

Alexis Ledru
(15311589/ope342)
University of Amsterdam
Amsterdam, The Netherlands
alexis.ledru@student.uva.nl

Jonathan Meeng
(2737479/JMG640)
University of Amsterdam
Amsterdam, The Netherlands
jonathan.meeng@student.uva.nl
j.j.meeng@student.vu.nl

River Vaudrin
(11877154/rva224)
University of Amsterdam
Amsterdam, The Netherlands
river.vaudrin@student.uva.nl

1 INTRODUCTION

Evolutionary computing utilises the principles of Darwinian evolution to optimise solutions to certain problems. Darwinian evolution stipulates that the best solution created by mutation and/or reproduction is selected. To quantify the best solution, a fitness function is designed and evaluated on all created solutions. In this paper, different evolutionary algorithms (EAs) are developed, applied and compared to play EvoMan. EvoMan is a Python Framework, which is designed to test and/or develop optimisation algorithms [1]. The fitness function is essential as it quantifies the quality of a strategy.

In this research, the behavioural difference between two genetic algorithms is optimised for two unique fitness functions, one that mainly prioritises preserving player health and one that mainly prioritises dealing damage to enemy health. We hypothesise that if player health is prioritised the optimal strategy will be more defensive, and conversely, when enemy health is prioritised, the strategy is more offensive.

2 METHODS

2.1 Experimental-Setup

Our methodology can be condensed to the following three steps: (1) sweeping the parameter space of the fitness function (w.r.t the weights that prioritise player/enemy health) to find two unique configurations of the fitness function; (2) optimizing our genetic algorithm [2] with these two configurations of the fitness function; (3) conducting a behavioural analysis based on game data that we extracted from the player controller. We decided to conduct our study on MetalMan, BubbleMan, and QuickMan. MetalMan is defeated by simply dodging and shooting, while BubbleMan and QuickMan have distinct attack abilities and tracking behaviour. In the case of BubbleMan, an enemy that has to be beaten underwater, the game's physics are altered to mimic these conditions. For the parameter sweep, or sensitivity analysis, we adjusted α and γ in increments of 0.1, with the constraint $\alpha + \gamma = 1$, and calculated the avg. mean fitness of the population at the final generation over three trials for each enemy. This allowed us to get a sense of how the algorithm performed with each combination of α/γ. After we picked two configurations, we optimized our genetic algorithm twice, for ten trials each, to find a total of twenty best strategies, ten defensive, and ten offensive. This was done for each of the three enemies. The final best solutions found for all enemies were used to extract behavioural measurements for each configuration. The metrics we extracted were player health, enemy health, gain, time, left, right, jumps, releases, shots, and accuracy. The accuracy is calculated as: accuracy = $\frac{100-e_e}{\text{shots}}$, where e_e represents the enemy's health. These metrics helped us establish the difference in behavioural patterns between the two configurations.

2.2 Genetic Algorithm

The genetic algorithm (GA), a well-known evolutionary algorithm, is designed to optimise a fitness score in a given search space over generations using the concepts of Darwin's evolution theory [3, Ch. 2]. GAs consider populations of solutions rather than a single solution, meaning the entire population should evolve over time rather than an individual. Typically, a GA requires at least a genetic representation and a fitness function. Moreover, it needs the ability to evolve its next-generation solutions to a better set of solutions with a higher fitness. This can be achieved by randomly mutating a solution, or by selecting parent solutions and applying crossover to create new solutions. Consequently, some old solutions should be discarded based on certain criteria (survivor selection). This allows the population to evolve but keeps its size fixed. A GA can be terminated at any given moment in time to evaluate the current best solution to the given problem. Note that GAs are stochastic; there is an element of randomness in the algorithm.

2.2.1 ***Fitness.*** The default fitness function of the EvoMan framework is: fitness = $\gamma * (100 - e_e) + \alpha * e_p - \log t$, in which e_e and e_p are the energy measures of enemy and player at the end of a game, respectively, varying from 0 to 100; t is number of time steps it took to complete the game; and γ and α are constants with default values set to 0.9 and 0.1 respectively. As mentioned, γ and α can be seen as a trade-off between preserving player health and dealing damage to enemy health, respectively.

2.2.2 ***Representation.*** The individual solutions in our population use a real-valued representation, where every individual in the population is a solution to the EvoMan game. A solution is an array of floating-point numbers that represent the weights and biases for a neural network (NN) that is attached to the player controller. Here, the genotype is represented as the weights and biases of the NN and the phenotype is the NN itself. The array contains 265 float values between -1 and 1. This is due to the structure of our fully connected NN: 20 input nodes, 10 hidden layer nodes, and 5 output nodes. Additionally, all hidden nodes and output nodes have biases (N=15). The inputs of the NN are the 16 projectile distances, player direction, enemy direction and enemy distance. These values are extracted and put through the NN at every frame of the game. The NN returns 5 float values that are rounded to 0 (do nothing) or 1 (do something). The output values are walk left, walk right, jump, shoot, and release of the jump.

2.2.3 ***Tournament Selection.*** Given the EvoMan framework, we are dealing with a game strategy optimisation problem which means that the pool of solutions can be very large. Additionally, we are constrained to only using player health, enemy health and game duration to compute a fitness function. Such fitness functions cannot provide a good quantitative measurement of the quality of gameplay. Tournament selection can solve both of these problems. This selection method picks k random individuals from the population, called a tournament, and ranks and compares their fitness scores. Then, λ best individuals are chosen within a tournament to proceed to the next generation and reproduce offspring. Tournament selection circumvents the mentioned complications since it does not require knowledge of the entire population and it ranks the solutions in the population given their fitness score rather than comparing the absolute fitness scores of each solution [4, 5].

2.2.4 ***Blend Crossover.*** Optimal combinations can be found by examining combinations of many parameters. Therefore, it is interesting to explore a search space that is larger than the limited search space between the parents' values. Blend crossover is often applied to focus on exploration rather than exploitation of the search space. Moreover, this method is able to escape local optima which allows to explore more of the search space instead of converging at these values. Offspring can be created by computing $\gamma = (1 - 2\alpha)u - \alpha$ where u is a randomly drawn value from $[0, 1]$.

Parameter	Default Value	Explored Space
n_population	75	-
generations	20	-
γ (fitness)	0.9	[0.1 - 0.9]
α (fitness)	0.1	[0.9 - 0.1]
λ (selection)	50	[10, 25, 50]
k (selection)	8	[2, 4, 8]
α (crossover)	0.5	-
mutation step-size	0.01	[0.01, 0.05, 0.1]

Table 1: Genetic Algorithm: parameters and their default value.

Then, calculate new values for the child using $z_i = (1-\gamma)x_i + \gamma y_i$ [6].

2.2.5 ***Nonuniform Mutation.*** It is important to determine the preferred balance between the complexity of mutation and the duration of mutation computation. Since several parameters are optimised, the computations should not be as expensive as a self-adaptive mutation, but still more sophisticated than a straightforward uniform mutation. Logically, the most suited mutation method for this problem is a nonuniform mutation which is not computationally heavy yet somewhat more complex in computing mutations. It draws a random value from a Gaussian distribution ($p(\Delta x_i) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(\Delta x_i - \epsilon)^2}{2\sigma^2}}$) with mean zero and user-defined standard deviation σ (mutation step size). As a consequence, mutations are more likely to provide small deviations which explore the local search space around the mutated parent. Nevertheless, huge mutations can occur since the probability distribution never reaches zero; exploring the search space on a more global level.

2.2.6 ***Replace Worst.*** Individuals that move forward to the next generation are selected using the replace worst method. Given this problem's large population size, rapidly increasing the mean population fitness is beneficial. In addition, the replacement procedure is desired to not be time-consuming and/or computationally expensive. By simply eliminating the λ worst children, the mean fitness value can rise very quickly using fast and cheap computation. On the contrary, the GA may converge prematurely due to the hyperfocus on the population portion with the highest fitness scores. However, this is negligible due to the large population size.

2.2.7 ***Parameter Tuning.*** Parameter tuning is a crucial step when constructing a GA, as it can have a significant impact on their performance and convergence behaviour. We implemented a grid search approach to find the optimal parameter values for λ (total number of parents to select per generation), k (tournament size), and mutation step-size. In a grid search, the user defines a set of potential values for each parameter of interest, creating a "grid" or a multi-dimensional space of possible parameter combinations. This is a computationally expensive task, hence only three parameters are tuned for a limited space. The remaining parameters were selected based on results from our experiments and computational cost, and can be seen in Table 1. To find the optimal parameter values, we let each parameter configuration in the grid play against the three enemies for three trials. The configuration that scored the highest mean fitness at the final generation over all enemies and trials, were selected as the most optimal parameters, these were $\lambda = 50$, $k = 8$, mutation step-size = 0.01.

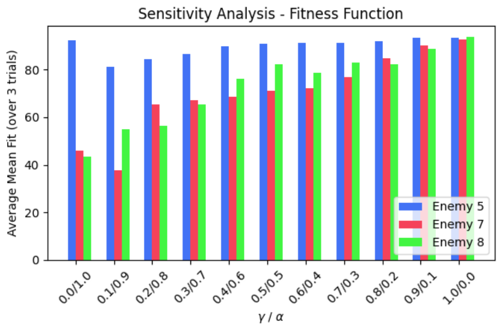

Figure 1: Results of the Sensitivity Analysis with respect to the fitness function's γ and α.

3 RESULTS

3.1 Sensitivity Analysis

In Figure 1, we can see the avg. mean fitness for eleven unique configurations of γ and α for each of the three enemies. These results show that optimisation trials where draining enemy health is prioritised ($\gamma > \alpha$) generally produce a final generation of higher-fitness individuals compared to trials where preserving player health ($\gamma < \alpha$) is prioritised. However, the results of enemy 5 prove an exception to this rule, as the configuration where strictly player health is prioritised ($\gamma = 0.0; \alpha = 1.0$) can generate final populations that have a similar avg. mean fitness as the trials where $\gamma > \alpha$. This is due to the fact that enemy 5 is unable to track the player, it is only able to throw objects to the player. This means that optimisation trials where $\gamma < \alpha$ are able to learn strategies wherein no shot is fired, and all objects are dodged until the game time expires, which results in a fitness score of $100 - \log t$ (t=3000 max game time). We decided to choose the configurations $\gamma = 0.9; \alpha = 0.1$ and $\gamma = 0.1; \alpha = 0.9$, which we coined *"The Striker"* and *"The Ninja"*, respectively. These configurations, which are on opposite sides of the spectrum, will allow us to examine the difference in learned behaviour between an offensive and defensive fitness function.

3.2 The Striker: $\gamma = 0.9; \alpha = 0.1$:

In Figure 2, the training progress of the two configurations is depicted for each enemy over ten optimization trials. For the *Striker* configurations, these results show that the algorithm tends to converge within ten generations to a population mean fitness that is close to the max fitness individual in the population. Interestingly, the avg. max fitness is close to the maximum achievable score at initialization, which means that there tends to be an individual in the population that is initialized with a score close to the maximum fitness. For all three enemies, the configuration is able to consistently find solutions that can beat their respective enemies. This is also confirmed when we look at Figure 3, we can see that the best *Striker* solutions are all able to beat their opponent, as the *Striker* solutions never score an individual gain below 0.

3.3 The Ninja: $\gamma = 0.1; \alpha = 0.9$:

The training progress graphs of the *Ninja* are noticeably different from those of the *Striker*. The results show that the *Ninja* parameter values tend to converge within ten generations to a population

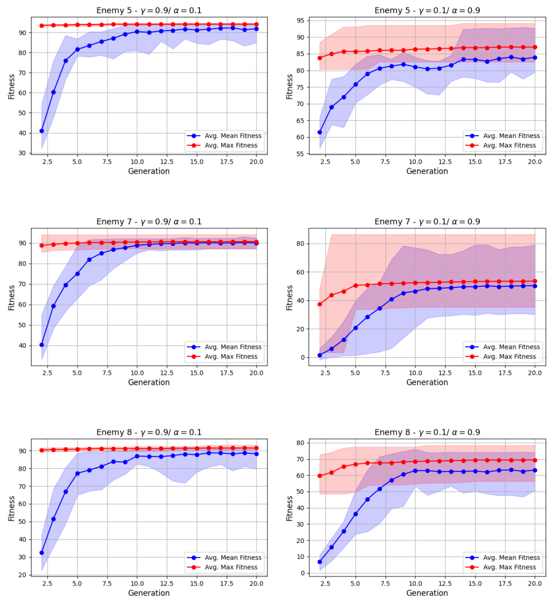

Figure 2: Results of training progress for each enemy, for each configuration: *The Striker* (left), *The Ninja* (right). The bounds show the min/max mean fitness for each generation and trial.

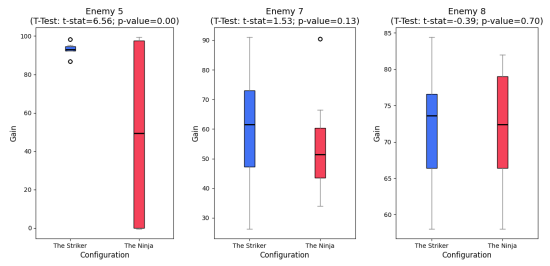

Figure 3: Results of the best solutions (of 10 trials) of both configurations playing against their respective enemy (5 times).

mean fitness that is close to the maximum fitness individual in the population. In contrast to the *Striker*, the *Ninja* trials often converge to a population mean fitness that is considerably lower (for each enemy). Presumably, because it is more difficult to preserve health than to deal damage, and due to the *Ninja* prioritising health preservation, solutions are more likely to score lower fitness evaluations. This would also explain the difference in the avg. max fitness and the large variance in avg. max fitness/avg. mean fitness between trials. As the possible solution space is smaller for the *Ninja*, the algorithm is more likely to be stuck in local optima and is highly dependent on randomly initialised solutions or mutations. Note that parameter tuning was conducted with $\gamma = 0.9$ and $\alpha = 0.1$, this could also provide a reason as to why the *Ninja* performs worse than the *Striker*. Figure 3 shows that the best *Ninja* solutions are also all able to beat their opponent. When comparing the mean gains of both configurations, there is only a significant difference found for enemy 5 (p-value $\leq$ 0.05). This is due to the fact that

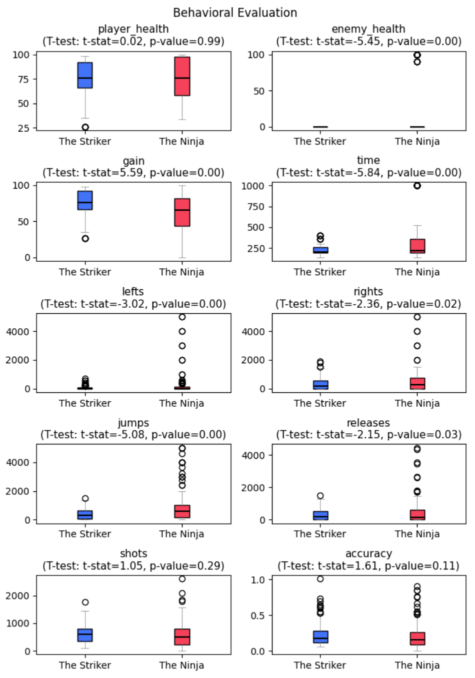

Figure 4: Results of the behavioral metrics for each configuration, for all three enemies combined.

some of the optimal *Ninja* solutions for enemy 5 are those where the player is able to dodge all of the incoming objects until the game time runs out.

3.4 Behavioral Comparison

In Figure 4, the results of the behavioural evaluation between the *Striker* and the *Ninja* can be seen. These results were obtained by letting each best solution over ten trials and three enemies play against their respective enemies five times. The results show that the median behaviour for each metric is similar, whereas the mean behaviour is not. For *enemy health*, *shots*, and *accuracy*, no significant difference was found between the means of both configurations (p-value $\geq$ 0.05). This means that with respect to the other behavioural metrics, there is a significant difference in behaviour. The results show that the *Ninja* tends to move around more, shoot less accurately but more (or not at all in the case of enemy 5), and play longer games. The *Striker* seems to be overall more efficient, in terms of movement and shots in beating the enemy.

4 CONCLUSION

In summary, both the *Striker* and *Ninja* configurations are able to consistently find solutions that are able to beat their respective enemy. However, the *Striker* is able to consistently produce populations that are full of high-fitness individuals, whereas the *Ninja* is more dependent on the stochastic processes of the GA. In terms of the difference in the behaviour between the two configurations, the *Striker* is more efficient in beating their enemy (more offensive), whilst the *Ninja* exhibits more dodging behaviour and tends to waste time (more defensive).

REFERENCES

[1] K. da Silva Miras de Araujo and F. O. de Franca, "Evolving a generalized strategy for an action-platformer video game framework," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1303–1310, 2016.

[2] G. 51, "https://github.com/am0nke/evolutionarycomputing_group51," 2023.

[3] S. Sivanandam, S. Deepa, S. Sivanandam, and S. Deepa, *Genetic algorithms*. Springer, 2008.

[4] T. Bäck, "Generalized convergence models for tournament-and (mu, lambda)-selection," in *Proceedings of the 6th International Conference on Genetic Algorithms*, pp. 2–8, 1995.

[5] T. Blickle and L. Thiele, "A comparison of selection schemes used in evolutionary algorithms," *Evolutionary Computation*, vol. 4, no. 4, pp. 361–394, 1996.

[6] L. J. Eshelman and J. D. Schaffer, "Real-coded genetic algorithms and interval-schemata," in *Foundations of genetic algorithms*, vol. 2, pp. 187–202, Elsevier, 1993.