Evolutionary Computing Task 1: specialist agent[*]

Team 21[†]

Cormac Berkery
University of Amsterdam
cormac.berkery@student.uva.nl

Martin Frassek
University of Amsterdam
martin.frassek@student.uva.nl

Koen Greuell
University of Amsterdam
koen.greuell@student.uva.nl

Roland Hoffmann
University of Amsterdam
roland.hoffmann@student.uva.nl

CCS CONCEPTS

• **General and reference → Experimentation**; *Design*; *Performance*; • **Theory of computation → Evolutionary algorithms**; • **Computing methodologies → Neural networks**; *Genetic programming*;

KEYWORDS

Evoman, NEAT, Genetic algorithm, Island model

ACM Reference Format:
Cormac Berkery, Martin Frassek, Koen Greuell, and Roland Hoffmann. 2019. Evolutionary Computing Task 1: specialist agent: Team 21. In *Proceedings of VU Evolutionary Computing (GECCO '19).* ACM, New York, NY, USA, 4 pages. https://doi.org/

[*]Produces the permission block, and copyright information
[†]The full version of the author's guide is available as `acmart.pdf` document

GECCO '19, October 4, 2019, Amsterdam, Netherlands
2019.

1 EVOMAN

1.1 Introduction

In this work evolutionary computation was used to optimize a player strategy for the video game Evoman[1]. Evoman is a two dimensional video game consisting of fights with eight different enemies in one-on-one tournaments. The game has been specifically designed to facilitate the development of evolutionary algorithms. It features 20 sensors, the majority of which provide information on the various projectiles (e.g. their height and distance from the player) as well as those regarding the character positions and directions. Players can perform five actions at any time: move left, move right, jump, interrupt jump and shoot. We researched and implemented three evolutionary algorithms in Python to challenge enemies 5, 6 and 7 of Evoman. Miras[2] showed that enemy 5 is easy to beat while enemy 6 is less trivial. Enemy 7 is of intermediate level and due to changed physics forces the player to develop a strategy involving the interruption of jumps. We will first describe a genetic algorithm (GA) evolving the weights of edges in a bipartite neural network with a single hidden layer. Secondly, a genetic algorithm with evolving edge weights and additionally generated edges and nodes will be described (NEAT). For the bipartite network we also developed an algorithms evolving multiple populations in parallel with migrations between populations, known as the Island Model. Based on Miras[2] we expected that NEAT would provide good solutions faster than the genetic algorithm with a fixed network. To enable fair comparison between the algorithms, all were run for 100 generations with 100 individuals.

2 GENETIC ALGORITHM

2.1 Introduction

The GA was developed by the authors, and was used to evolve the weights of a complete bipartite neural network in order to develop a player strategy for Evoman. At each time step, the sensor values are fed to the network and a decision regarding the next move is output and executed. At the end of the game, the standard fitness evaluation formula stated in Miras[2] is used to determine relative performance of the network.

2.2 Parameters

The network evolved by the genetic algorithm featured a hidden layer of 50 neurons connecting the 20 sensors with the five possible actions. The nodes had weights between -1 and 1 which were initialized randomly from a uniform distribution. Fitness scores were determined by letting the population of 100 individuals play Evoman, using the default fitness function described in the Evoman paper[2]:

$$f = 0.9 \cdot (100 - \text{enemy life}) + 0.1 \cdot \text{player life} - \log \text{time}$$

The choice of 100 individuals was made to obtain reasonable variation within limited computational time. Parent selection randomly matched 50 pairs of individuals and sent the fitter of each pair to the mating pool to stimulate exploration of the fitness landscape, resulting in 50 parents. In the mating pool parents were randomly sorted into groups of two which produced two children. We chose random intermediate recombination of the parents edge weights to determine the weights of the child, as done by Miras[2]. This lead to some variance in weights of the edges of the children within the range of the weights of successful parents. Next, each edge of the children was creep mutated with a probability of 20% (like in Miras[2]) in order to explore values outside the domain of the parents genome. If an edge was mutated, a random normally distributed number was generated with a mean of zero and a standard deviation of 1 until the number could be added to the edge weight, while keeping the edge weight bounded between -1 and 1. The 100 individuals with the best fitness scores were passed on to the next generation in order to quickly increase the fitness.

Table 1: Genetic Algorithm Parameters

Parameter	Value
children per generation	50
crossover probability	100%
crossover type	random intermediate
mutation probability	20%
mutation distribution	normal

3 NEAT

3.1 Introduction

The NeuroEvolution of Augmenting Topologies (NEAT) algorithm is an evolutionary algorithm which extends the GA. Not only does the NEAT algorithm change the weights of the edges between the nodes, but nodes and edges can also be added and removed. An immediate implication is that the structure of the neural network is not fixed and more flexible than GA. Direct edges between the sensors and actions can also be created without an intermediate hidden node.

We have used the Python package NEAT-python to implement the NEAT algorithm for Evoman.

3.2 Parameters

We have used several parameters to tune the NEAT algorithm, which will be discussed in this section.

The neural network is initialized with the following properties: A layer with 50 hidden nodes is generated and each of the nodes is connected to all sensors and all actions, comparable to GA. The weights of the edges are then picked randomly from a normal distribution with mean 0 and standard deviation 1. Furthermore each node is initialized with a bias value which is also sampled from a normal distribution with mean 0 and standard deviation 1.

During the run different mutations can occur with a certain mutation probability, and some mutations also have a certain mutation power. The mutation probability is the probability that a mutation occurs. The mutation is a number which is added to the current value which is sampled from a normal distribution with mean 0 and another parameter, mutation power, as the standard deviation. The following mutations can occur; node bias change, add edge, remove edge, add node, remove node and edge weight change. For our experiments we have used the values shown in table 3.

Note that the bias and weight are bounded such that are always in the range [-30, 30]. Finally each genome has a 20% chance of

Table 2: Model parameters for the NEAT algorithm.

Mutation	Mutate rate	Mutate power
node bias	0.7	0.5
edge weight	0.8	0.5
add/remove edge	0.5	-
add/remove node	0.2	-

dying randomly each generation in order to increase the diversity within the group.

4 ISLAND MODEL

4.1 Introduction

The Island model is a meta model of evolutionary algorithms which allows for multiple sub populations to evolve in parallel, with the goal of having each group follow a different search trajectory. By exploring a larger area of the search space this method has the potential to produce better solutions while requiring fewer iterations.

Individuals are periodically exchanged between sub-populations, giving us two new hyper-parameters, the migration frequency and migration size. Islands are randomly paired during each migration, and the fittest individuals are copied from one island and put in place of the weakest individuals on the other island.

Empirically, evolving multiple populations tends to outperform single populations for problems that can be divided into disjoint sub problems. Beating the Evoman game could be seen as such a problem, where the attack of the enemies can be viewed as set of sequences. As an example, one population could become proficient in jumping over the first set of bullets and dealing damage, yet perform poorly in another phase - migration can be a way to combine these sub-strategies.

4.2 Parameters

Our Island model was based on four different islands. Each island contained its own population of 25 individuals and each individual had a neural network with 50 hidden nodes, identical to the GA. With a a total of 100 individuals a comparison with the other algorithms is still feasible. The new parameters; migration size and migration frequency were set to 2 individuals and 10 generations respectively. The migration parameters were balanced such that we do not homogenize the islands by exchanging too often, but still combine diverse solutions.

5 RESULTS

We have implemented the above algorithms in Python 3.7 and ran each model ten times for each of the three enemies to find the following results[1]. As expected all algorithms performed best on enemy number 5 and worst on enemy number 6. The best fitness in generation 100 is similar for the Island model and standard GA (see figure 1), while the mean fitness in generation 100 for enemy 7 is slightly higher for the island model (see figure 2). NEAT outperforms the other algorithms, except at enemy 5, where all algorithms manage to obtain an almost optimal solution.

[1]See GitHub repository for code and results.

Table 3: Results of the implemented algorithms. Averaged best fitness in the last generation over 10 runs.

Enemy	5	6	7
GA	91	18	63
Island	91	16	59
NEAT	93	66	91

The variation of the mean fitness is highest for the NEAT algorithm (see figures 3, 4 and 5), as is the variation in the best fitness for NEAT for enemy 6, while the variation for the best fitness is low for NEAT for enemy 5 and 7 as it has reached an almost optimum fitness value of higher than 90. The fitness improvement per generation seems to be negatively correlated with the generations such that the majority of improvement is reached at the beginning of the evolution. Also noticeable is the variation in the best fitness which only increases in the GA and Island model as it removes only the individuals with the lowest fitness, while in the NEAT model all individuals have a 20% chance of being removed.

The fitness in the genetic algorithm improves quite slowly as the fitness of an individual is only determined once upon creation. We analyzed the variation in fitness score of a single untrained individual by running it 20 times against enemy 7, finding a mean fitness score of 12.95 with a standard deviation of 8.15. The standard deviation was higher for this single individual then for 20 unique untrained individuals tested against enemy number 7 (Standard deviation: 7.50; Mean: 17.04). This suggests that a (μ, λ) Selection would be more suitable for this algorithm as is prevents individuals who have obtained a high fitness value by chance from siring a lot of less fit offspring. Alternatively the fitness of an individual could be reevaluated after each generation, although coming at the cost of higher computation time.

As expected we see that all algorithms show significant growth in the first 10 generations and then flatten out. Dependent on the enemy and algorithm this effect is stronger or weaker, but it is very visible in figures 3, 4 and 5.

Even though all algorithms perform very well against enemy 5, this enemy proved to take the longest CPU time to train. The other enemies could be trained within a few hours or shorter for all algorithms, but enemy 5 could easily take half a day. On top of that we noticed that the NEAT algorithm was significantly slower than the other two algorithms by a factor 2-3.

Compared to the results obtained Miras[2] the NEAT algorithm implemented in this work performed similarly well to the one displayed there. Here it was also corroborated that NEAT can outperform other algorithms like the GA.

6 CONCLUSION

With the same population size and same amount of generations the genetic algorithm evolving a NEAT network outperforms the standard genetic algorithm evolving a fixed topology bipartite network and the island model variant of this algorithm by consistently producing a better mean fitness and a better maximum fitness for hard problems such as enemy 6 and 7. Expanding the standard GA model with separated populations on islands hardly improves its

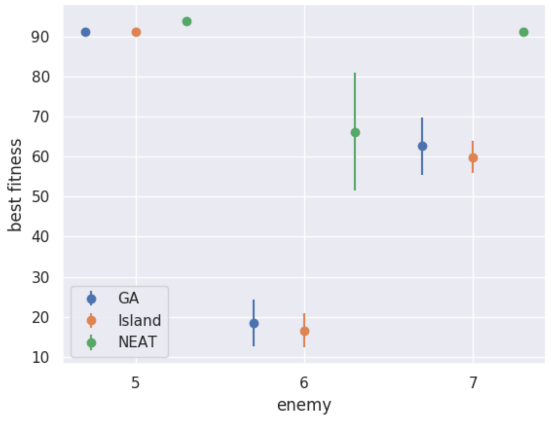

Figure 1: Best fitness score of the genetic algorithms in the 100th generation, showing the mean and standard deviation of 10 runs.

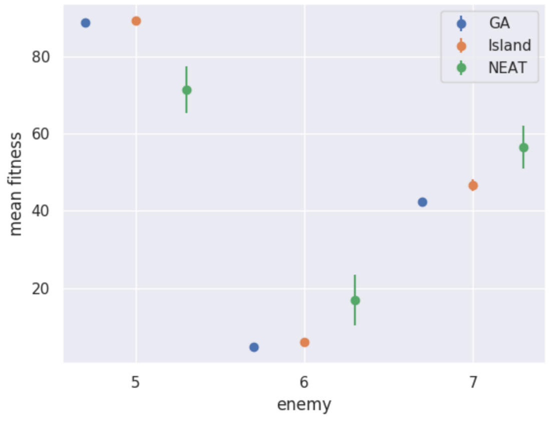

Figure 2: Mean fitness score of the genetic algorithms in the 100th generation, showing the mean and standard deviation of 10 runs.

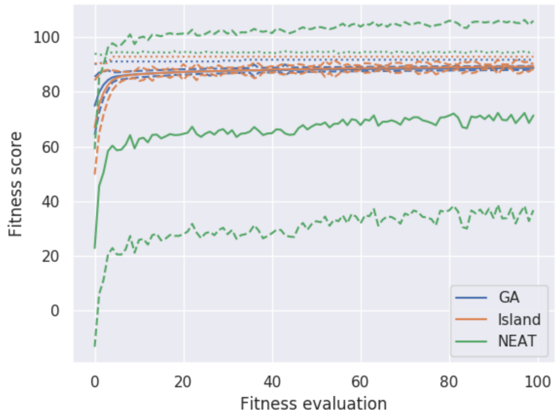

Figure 3: Mean mean (solid line), mean mean ± standard deviation (dashed line) and mean maximum (dotted line) fitness of 10 runs of the genetic algorithms against enemy 5.

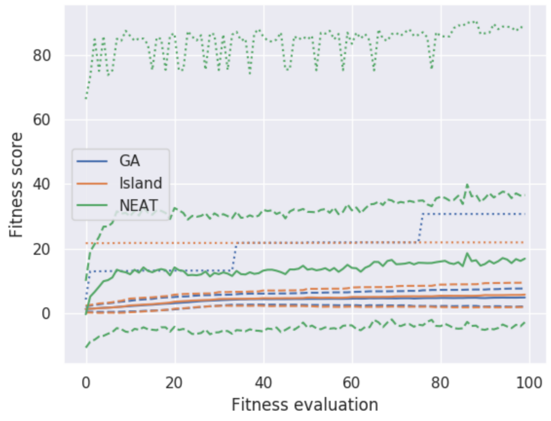

Figure 4: Mean mean (solid line), mean mean ± standard deviation (dashed line) and mean maximum (dotted line) fitness of 10 runs of the genetic algorithms against enemy 6.

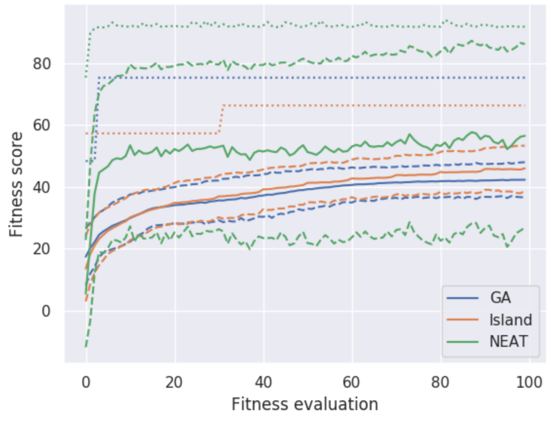

Figure 5: Mean mean (solid line), mean mean ± standard deviation (dashed line) and mean maximum (dotted line) fitness of 10 runs of the genetic algorithms against enemy 7.

performance. NEAT is capable of solving the problems in less than 10 generations, after which the mean fitness slowly continues to improve, while the best solution remains stochastically stable.

REFERENCES

[1] Miras, Karine, F. O. (2016a). An electronic-game framework for evaluating coevolutionary algorithms. arXiv preprint arXiv:1604.00644.
[2] Miras, Karine De França, Fabricio. (2016). Evolving a generalized strategy for an action-platformer video game framework. 1303-1310. 10.1109/CEC.2016.7743938.