Evolutionary Computing - Assignment 2: Generalist agent

Team 88 - October 18, 2021

Nedim Azar (2728313)
Vrije Universiteit
Amsterdam, Netherlands
n.azar@student.vu.nl

Simone Montali (2741135)
Università di Bologna
Bologna, Italy
s.montali@student.vu.nl

Giuseppe Murro (2740494)
Università di Bologna
Bologna, Italy
g.murro@student.vu.nl

Martin Pucheu Avilés (2687903)
Vrije Universiteit
Amsterdam, Netherlands
m.i.pucheuaviles@student.vu.nl

ACM Reference Format:
Nedim Azar (2728313), Simone Montali (2741135), Giuseppe Murro (2740494), and Martin Pucheu Avilés (2687903). 2021. Evolutionary Computing - Assignment 2: Generalist agent: Team 88 - October 18, 2021. In *Proceedings of X400111 (Evolutionary Computing).* , **??** pages. https://doi.org/

1 Introduction

1.1 Evolutionary computing

The term evolutionary computing refers to a field that is concerned with designing, applying, and studying evolutionary concepts in computing. We could consider it as the link between problem-solving and biological evolution. By applying these evolutionary concepts to candidate solutions of a problem (which, in EC, are equivalent to Individuals), we can find the optimum efficiently.

1.2 The EvoMan framework

The EvoMan framework [19] was created with the purpose of testing the performance of Evolutionary Algorithms in a standard way: it consists of a game in which a player has to fight against an enemy that shoots, jumps and moves. The framework provides 8 different enemies, each having different behavior and ways of winning. Having 20 sensors as input, we would like to find a *neural network* that maps these values to an optimal action for the player, having a fitness that is represented by the score of the player in the game.

1.3 Research question

The difference between multi-objective and single objective optimization is very important to consider during Evolutionary Algorithms design: it is not possible to have a single solution which simultaneously optimizes all objectives, so it is necessary to settle for suboptimal solutions, which are generic enough to return good results for every objective. The most straightforward approach to tackle this problem is the one seen in the EvoMan framework: we can create a multi-objective function defined as

$$f = avg([f_1, ..., f_n]) - \sigma([f_1, ..., f_n]) \tag{1}$$

with f_i being the fitness obtained with enemy i. Over the past decade, though, a number of multi-objective algorithms have been suggested [4, 5, 10, 11]. Most of these use non-dominated sorting and sharing, which have been criticized for their computational complexity. Kalyanmoy Deb et. al proposed a new algorithm for multi-objective optimization, namely *Non-dominated Sorting GA-II*[6], which has a lower computational complexity and has been proved to outperform most of the algorithms which are available in literature, including *PAES* [15], another elitist algorithm. In this research, we'll want to compare the performances of this multi-objective aimed algorithm to a single-objective evolutionary algorithm optimizing function 1. Both the algorithms use self-adaptive mutation [21] and blend crossover [8]. To ensure diversity of the individuals, the single-objective algorithm exploits the *island model* [22]. The results will prove how a single-objective optimization algorithm (adapted to multi-objective using function 1) compares to an algorithm that was specifically made for multi-objective tasks.

2 Methods

2.1 Two different approaches

As aforementioned, the first approach will consist in a custom-tailored Genetic Algorithm which optimizes the EvoMan default multi-objective fitness function, while the second will be variation of the same algorithm implementing *Non-dominated Sorting GA-II* [6] for the selection of survivors, ranking solutions in *non-dominated fronts* for a set of objectives.

2.2 Approach 1: island model and default objective function

2.2.1 Overview. In this approach, we will try to maximize the fitness function seen in 1.3, namely, the subtraction between the standard deviation and the average of the single-objective fitnesses.

2.2.2 Representation, genotypes and phenotypes. Choosing the correct representation for a problem is the most important task in the design of an evolutionary algorithm. As mentioned in section 1.2, we would like to obtain an individual representing an optimal neural network that is able to map sensors' inputs to a player action. A straightforward approach to represent a NN would be using a real-valued array containing the weights and biases for the neural network. We could keep this array flat, then reshape it when needed, as we know the network structure. In fact, to convert the genotype to a phenotype (the NN), it's sufficient to copy the values representing the biases, then reshaping the remaining array with `numpy` [14] to represent the layers' weights. The network structure is a conventional one: the layers are fully connected, and a *sigmoid activation function* [20] is used in-between. The network structure was designed as requested by the EvoMan competition: 20 input nodes, 10 hidden nodes, 5 output nodes.

2.2.3 Reproduction and mutation. Reproduction and mutation are crucial steps for a good exploration of the search space. The first thing that is needed to obtain a good offspring is applying what is known as *mating selection* (seen in section 2.2.4). Then, two of the chosen individuals are cloned into the offspring, and blend crossover is applied with a probability p_{cx}. This type of crossover was introduced in [9] to generate offspring in a region that is bigger than the n-dimensional rectangle spanned by the parents. This allows for a more thorough exploration of the search space, moving each attribute of the genome of parent 1 by a quantity $\gamma = (1 - 2\alpha)u - \alpha$ where u is a random number between 0 and 1, and $1-\lambda$ for parent 2. Mutation is the other important operation that allows us to explore the search space: each individual has a probability p_{mut} of mutating, and each attribute in this individual mutates with probability p_{indmut}. To achieve a non-uniform mutation, we implemented what is called self-adaptive mutation. This type of mutation sees a varying σ value, which is added to the genome of an individual and evolves together with it. Every attribute that has to mutate will then mutate by a quantity $x_i' = x_i + \sigma' \cdot N_i(0, 1)$. To further investigate how σ is updated, see section 2.2.6.

2.2.4 Selection. While variation operators strive for exploration, selection is concerned with exploitation. It allows us to select good individuals in the population, allowing them to survive and mate. We can talk about two types of selection: *mating selection* and *survival selection.* The first selects which parents will be able to reproduce, while the latter decide which individuals to replace (since we're maintaining a fixed population size, we can talk about replacement). To perform *mating selection*, we used a tournament selection. This type of selection allows us to only consider the fitnesses of a small subset of individuals, then select the best among these. What happens is that we extract t_s random elements from the population, then pick the best one. We repeat this process for k times, obtaining, consequently, k individuals. To perform survival selection, two main choices could be taken: the first is $(\mu + \lambda)$-selection, which considers both the present population and the offspring in the selection of the survivors. The second approach is (μ, λ)-selection, where all the parents are discarded in spite of a sometimes bigger offspring. As seen in [7] at section 5.3.2, the latter is usually preferred, as it's better at leaving local optima, forgetting outdated solutions, and the σ we're using for *self-adaptive mutation*. Because of this, we're performing (μ, λ)-selection to choose our offspring, using best selection: only the μ best individuals are kept, whereas the other $\lambda \cdot \mu$ are forgotten. Section 6.2 of [7] suggests a λ between 4 and 7, and tests led us to choose $\lambda = 7$.

2.2.5 Speciation. Parallel Genetic Algorithms have often been reported to yield better performance than Genetic Algorithms which use a single large panmictic population [22]. For this reason, while still keeping the default multi-objective fitness function, this approach tried to maximize the performance by implementing this kind of model. The population is now divided into n_{isl} islands, resulting in a total number of individuals $n_{isl} \cdot pop_{isl}$. Every i_{mig} iterations, some individuals migrate from an island to another. The migrant selection was designed by merging the approaches seen in [16] and [13], taking the worst (fitness-wise) n_{mig} individuals on an island and migrating them to an *attractive island*. The attractiveness of islands is measured basing on the fitness gain they had in the last generation: $f_i - f_{i-1}$ with i being the current generation, and f being the average fitness of the individuals on the island.

2.2.6 Parameter tuning and control. Parameter setting is a crucial part of every Artificial Intelligence algorithm, and Genetic Algorithms make no exception. We have to distinguish between *parameter control* and *parameter tuning*: the latter happens *offline*, before a run, while the first modifies the parameters continuously during a run. To perform parameter tuning, we exploited a Python library that allows users to explore the hyperparameter space, searching for the highest performances: `hyperopt` [1]. To achieve faster testing, we integrated our hyperparameter testing into a distributed computing platform, Apache Spark [23], then set up a cluster of cloud VPS to run experiments parallelly. The set of parameters that achieved the best results is shown in table 1.

Parameter control was a concern too: we implemented a self-adaptive mutation step size. This consists in adding the mutation step size to the genome of an individual, making it part of evolution. Ideally, we'll want to have a high step size in the beginning (fixed to a default value of 1.0), to better explore the search space, then start exploitation later in the evolution. We mutate the step-size as seen in section 4.4.1 of [7]: $\sigma' = \sigma \cdot e^{\cdot N(0,\tau)}$

α_{cx}	p_{cx}	p_{mut}	p_{indmut}	pop_{isl}	n_{isl}	i_{mig}	n_{mig}	t_s
0.56	0.90	0.52	0.28	8	10	4	6	7

Table 1: Parameter values, see 2.2.3, 2.2.4, 2.2.5

2.3 Approach 2: multi-objective optimization and NSGA-II

2.3.1 Overview. In order to compare the performances between single objective and multi-objective optimizations, a second genetic algorithm starting from the same genetic representation, crossover operator (blend crossover), mutation (self-adaptive mutation) and mating selection (tournament) was built. The difference between the two genetic algorithms comes in the implementation of *Non-dominated Sorting GA-II* [6] for the survivor selection. This elitist genetic algorithm incorporates the concepts of *multi-objective optimization*, *non-dominated fronts* and *crowding distance* to sort the population and guide the selection process towards a uniformly spread out *Pareto-optimal front*. For the parameter tuning, same methodology from approach 1 was used, ending with:

α_{cx}	p_{cx}	p_{mut}	p_{indmut}	t_s
0.44	0.75	0.66	0.7	6

Table 2: Parameter values, see 2.2.3, 2.2.4

2.3.2 Multi-objective optimization, Dominance and Pareto Front. In a *multi-objective optimization problem (MOP)*, every solution is evaluated by its performance in a group of possibly conflicting objectives. As evolutionary algorithms are population-based, they are well suited to find a diverse set of *high-quality solutions* even for a set of conflicting objectives. Here, the concepts of *Dominance* and *Pareto Optimality* acquire a significant relevance, as they define the characteristics that these solutions should have. We say that, for a set of objectives and different solutions, one solution dominates the other if its score is at least as high (for a maximization problem) for all objectives and strictly higher for at least one objective [7]. Then, if the problem presents conflicting objectives, there are no solutions dominating all others, therefore our interest is towards solutions that are not dominated by any other, called *non-dominated* solutions. The set of all non-dominated solutions is called the *Pareto Front*. For this particular problem, each objective function was taken to be the maximization of the individual fitness for each enemy of the training set, as an agent capable of performing well against a variety of enemies was needed.

2.3.3 Crowding. As a method to preserve diversity in the population and to avoid the premature convergence to local optima, *crowding* was introduced by De Jong [3] and different variations and upgrades have been proposed since then [12, 17, 18]. The basic idea is to compare *similar individuals*, that will compete for a place in the population, and to define the survivors for the next generation according to some similarity metric.

2.3.4 Survivors Selection: NSGA-II. The *Non-dominated Sorting Genetic Algorithm-II* proposed by Deb et al.[6], is a $(\lambda + \mu)$ approach that *ranks* all the solutions after the offspring creation by first defining the set of *non-dominated fronts*. In order to accomplish this, all the solutions need to be compared with one another to find whether one dominates the other. The *first non-dominated front* is composed by all the solutions that are not dominated by any other, thus, these are the highest rank solutions and the first ones to be selected for the next generation. This procedure is repeated to find the next front, excluding the solutions from the previous front, until all fronts are found. The next generation is then progressively filled up with the individuals from the first fronts, until the size of the population is achieved. If the individuals from the last front don't fit in the next generation, the *crowding distance* between them is considered as a second selection criterion, choosing solutions which are *located* in a region with a lower number of solutions. Because of this, a *crowding distance metric* is defined as the average side length of the cuboid defined by its nearest neighbours in the same front. Therefore, solutions with a bigger *crowding distance metric* are desirable.

3 Results

Lower performances are expected in comparison with single-objective tasks: optimizing different objective functions with a single solution is way harder, especially when done with a small neural network like the one presented in 2.2.2. Nonetheless, the multi-objective oriented optimizations seen in approach 2 should perform better. Both approaches has been implemented from scratch, without using any existing library for EC. To train the generalist agents, two sets of enemies were chosen: the first set consisted in enemies $[1, 5, 6]$, while the second consisted in enemies $[2, 5, 8]$. Choosing bigger sets of enemies was proved to be inefficient during our experiments. We can measure the performance of the trained individuals through the multi-objective gain measure, consisting in the sum of the gains (difference of player life and enemy life $l_p - l_e$) against all the enemies.

3.1 Results obtained with approach 1

As shown in figure 1, the genetic optimization has seen a slow, yet constant increase in the average fitness of the population across the initial generations for both the enemy sets. We can notice that the fitness stops increasing around generation 14, meaning an asymptote was probably reached. We can notice that when training against the set $[2, 5, 8]$, the algorithm does so faster. After that, diversity drops and the average fitness gets closer to the maximum. When testing the individuals we obtained during optimization, we can see that the two enemy sets have similar behaviour, yet different performances: the enemy set containing $[1, 5, 6]$ proved itself to be more general, as it performed slightly better.

3.2 Results obtained with approach 2

Similar to the genetic optimization in approach 1, an increase of both average and maximum fitness through the generations for both enemy sets is observed. It's interesting to notice that while for the set $[2, 5, 8]$ the maximum fitness stops increasing soon in the evolutionary process (approximately at generation 4), it took longer for the set $[1, 5, 6]$ to reach the asymptote, approximately at generation 14. This could give us a hint about the difference between both sets of conflicting objectives, with the set $[1, 5, 6]$ being a complex set to find a high-performance solution with.

3.3 Best solution

The best solution (gain-wise) among both the algorithms, for all runs, was then tested against all the enemies, 5 times per enemy, and the average results and the percentage of games won out of 5 are shown in table 3. During these runs, information gain reached an average of +24.

Enemy	1	2	3	4	5	6	7	8
Player life	65.6	0.0	0.0	1.6	80.9	45.8	0.0	6.6
Enemy life	6.0	42.0	20.0	10.0	0.0	0.0	98.0	0.0
Won games	80%	0%	0%	40%	100%	100%	0%	100%

Table 3: Best individual trained on [1, 5, 6] using approach 2

3.4 Comparison

The NSGA-II implementation seen in approach 2 obtained slightly better performance, which was expected as it was designed to solve this kind of problems. We can notice that the two algorithms have similar behaviour during training, with the average fitness increasing faster for the first set of enemies. Approach 2 seems better at maintaining diversity: the gap between average and maximum fitness is larger. Taking a look at the statistical tests in figure 2, we notice that while the second enemy set has been proven to be highly significant ($p = 0.00$), the first set has a higher p-value equal to 0.52. If we compare the results with [2], we can see that the results obtained during our experiment were way higher: the best individual (seen in section 3.3) obtained a gain (measure described in 3) of +24, while the best individual presented in the paper stopped at −65. The paper even states that *enemies 1 and 3 were never beaten by any of the evolved strategies*, but as seen in table 3, enemy 1 was beaten multiple times in this experiment.

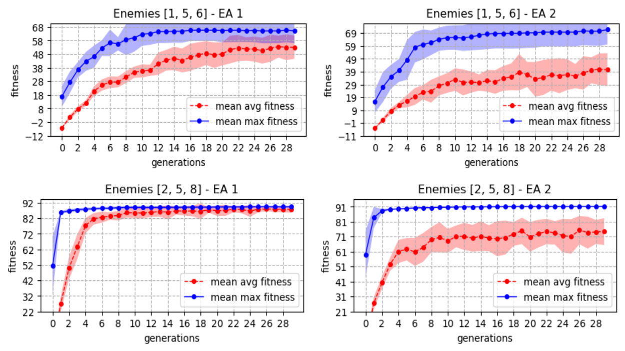

Figure 1: Line-plots of the fitness across the generations

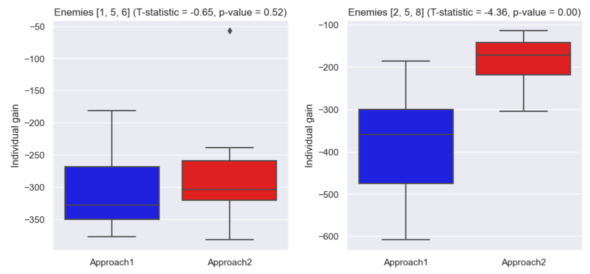

Figure 2: Box-plots of the gains for each of the 10 runs

4 Conclusions

As expected, the results for the multi-objective algorithm NSGA-II were better than the ones seen in approach 1. Multi-objective optimization proved itself to be a way harder task than single objective, as it's often impossible to find a single optimum able to maximize all the functions. Nevertheless, Evolutionary Algorithms are a good way of tackling the problems, and the number of applications for them will increase in the years to come.

4.0.1 Work division. Montali and Murro: approach 1. Pucheu Avilés and Azar: approach 2. All members: implementation, experimentation, comparison of results, and report writing.

References

[1] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, I–115–I–123.

[2] Karine da Silva Miras de Araujo and Fabrício Olivetti de Franca. 2016. Evolving a generalized strategy for an action-platformer video game framework. (2016), 1303-1310 pages. https://doi.org/10.1109/CEC.2016.7743938

[3] Kenneth Alan De Jong. 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph.D. Dissertation. USA. AAI7609381.

[4] Kalyanmoy Deb. 1999. Multi-objective genetic algorithms: Problem difficulties and construction of test problems. *Evolutionary computation* 7, 3 (1999), 205–230.

[5] Kalyanmoy Deb, Ram Bhushan Agrawal, et al. 1995. Simulated binary crossover for continuous search space. *Complex systems* 9, 2 (1995), 115–148.

[6] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *International conference on parallel problem solving from nature*. Springer, 849–858.

[7] A.E. Eiben and J.E. Smith. 2015. *Introduction to Evolutionary Computing*. Springer. https://doi.org/10.1007/978-3-662-44874-8 Gebeurtenis: 2nd edition.

[8] Larry J. Eshelman, Richard A. Caruana, and J. David Schaffer. 1989. Biases in the Crossover Landscape. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 10–19.

[9] Larry J. Eshelman and J. David Schaffer. 1993. Real-Coded Genetic Algorithms and Interval-Schemata. In *Foundations of Genetic Algorithms*, L. DARRELL WHITLEY (Ed.). Foundations of Genetic Algorithms, Vol. 2. Elsevier, 187–202. https://doi.org/10.1016/B978-0-08-094832-4.50018-0

[10] Carlos M Fonseca and Peter J Fleming. 1998. Multiobjective optimization and multiple constraint handling with evolutionary algorithms. II. Application example. *IEEE Transactions on systems, man, and cybernetics-Part A: Systems and humans* 28, 1 (1998), 38–47.

[11] Carlos M Fonseca, Peter J Fleming, et al. 1993. Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization.. In *Icga*, Vol. 93. Citeseer, 416–423.

[12] Severino F. Galan and Ole J. Mengshoel. 2010. Generalized Crowding for Genetic Algorithms. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO '10)*. Association for Computing Machinery, New York, NY, USA, 775–782. https://doi.org/10.1145/1830483.1830620

[13] Fujimura S. Gozali, A.A. 2019. DM-LIMGA: Dual Migration Localized Island Model Genetic Algorithm—a better diversity preserver island model. *Evol. Intel.* 12, 1 (2019), 527–539.

[14] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[15] Joshua D Knowles and David W Corne. 2000. Approximating the nondominated front using the Pareto archived evolution strategy. *Evolutionary computation* 8, 2 (2000), 149–172.

[16] Mohamed Kurdi. 2016. An effective new island model genetic algorithm for job shop scheduling problem. *Computers Operations Research* 67 (2016), 132–142. https://doi.org/10.1016/j.cor.2015.10.005

[17] R. Manner, Samir Mahfoud, and Samir W. Mahfoud. 1992. Crowding and Preselection Revisited. In *Parallel Problem Solving From Nature*. North-Holland, 27–36.

[18] Ole J. Mengshoel and David E. Goldberg. 2008. The Crowding Approach to Niching in Genetic Algorithms. *Evol. Comput.* 16, 3 (Sept. 2008), 315–354. https://doi.org/10.1162/evco.2008.16.3.315

[19] Karine Miras. 2020. EvoMan Framework. https://github.com/karinemiras/evoman_framework. (2020).

[20] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378* (2018).

[21] Jim Smith and Terence C Fogarty. 1996. Self adaptation of mutation rates in a steady state genetic algorithm. In *Proceedings of IEEE international conference on evolutionary computation*. IEEE, 318–323.

[22] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. 1999. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology* 7, 1 (1999), 33–47.

[23] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (nov 2016), 56–65. https://doi.org/10.1145/2934664