A novel distributed nature-inspired algorithm for solving optimization problems*

J. C. Felix-Saul[1], Mario García Valdez[1], and Juan J. Merelo Guervós[2]

[1] Department of Graduate Studies, Tijuana Institute of Technology, Tijuana, Mexico
[2] Department of Computer Architecture and Technology, Universidad de Granada, Granada, Spain

Abstract. Several bio-inspired algorithms use population evolution as analogies of na-ture. In this paper, we present an algorithm inspired by the biological Life-Cycle of animal species, which consists of several stages: birth, growth, re-production, and death. As in nature, we intend to execute all these stages in parallel and asynchronously on a population that evolves constantly. From the ground up, we designed the algorithm as a cloud-native solution using the cloud available resources to divide the processing workload, among several computers or running the algorithm as a cloud service. The algorithm works concurrently and asynchronously on a constantly evolving population, using different computers (or containers) independently, eliminating waiting times between processes. This algorithm seeks to imitate the natural life cycle, where new individuals are born at any moment and mature over time, where they age and suffer mutations throughout their lives. In reproduction, couples match by mutual attraction, where they may have offspring. Death can happen to everyone: from a newborn to an aged adult, where the individual's fitness will impact their longevity. As a proof-of-concept, we implemented the algo-rithm with Docker containers by solving the OneMax problem comparing it with a basic (sequential) GA algorithm, where it showed favorable and prom-ising results.

Keywords: Distributed Bioinspired Algorithms · Genetic Algorithms · Cloud Computing.

1 Introduction

Bio-inspired algorithms have been very successful when used to solve complex optimization problems, but as their complexity increases so does the computing power required. One strategy to address this processing need is to use distributed computing, or the resources available in the cloud to help find the solution. This strategy gives us elasticity to adjust (increase or reduce) the computing power to achieve a balance according to the nature of the problem.

One of the problems that we have identified in this research is that most bio-inspired algorithms are designed with a traditionally sequential perspective,

* Supported by TECNOLÓGICO NACIONAL DE MÉXICO (TecNM).

where each process must wait for the previous task to finish before continuing. Some architectures address this issue, what distinguishes us, is that our proposal presents an algorithm designed in its entirety as a native solution in the cloud, fully distributed, where its processes are executed in parallel and asynchronously. This technique makes it easy to scale the computing power according to the complexity required by the problem.

We designed our algorithm using observation, analysis, and abstraction in nature to identify what most species in the animal kingdom have in common, where individuals of a population that best adapt to the environment have greater chances of survival, reproduction, and improvement of the species. Imagining as if the task of writing these rules of evolution was in our hands, questioning how we could solve it?

Like nature, we identified what most species have in common in the life cycle. Our algorithm works on a constantly evolving population, experiencing the stages that all living beings undergo [7]: being born, growing up, reproducing, and dying, where each of these stages will be processes (of our algorithm) that randomly affect the population individuals, emulating the organic way.

In this research, we seek to demonstrate that it is possible to evolve a population of individuals, similar to a Genetic Algorithm (GA), using a distributed, parallel, and asynchronous methodology.

2 Description of the sections of the Paper

3 State of the art

Multiple citations are grouped [9, 3, 10, 5, 4, 1, 8].

3.1 Background

One of the publications with the most influence on our research proposes a cloud-native optimization framework, that can include multiple (population-based) algorithms [9]. In their research, they work with ideas such as those mentioned in Fig. 1.

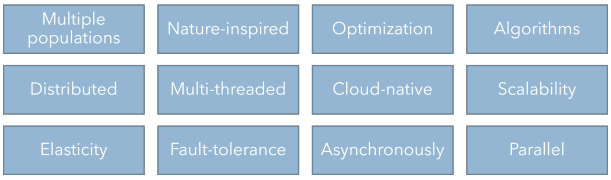

Fig. 1. These ideas planted the seed of thought in our current presentation.

4 Proposal

Our main inspiration for this algorithm was born from the study and observation of nature, where many successful optimization algorithms have previously found their own. Our focus was not on a single animal species but all of them, using general abstraction to identify what those species have in common.

In our analysis, we consider a broad animal spectrum, from bacteria to humans and lions, finding our inspiration on the biological Life Cycle of the animal species, which consists of several stages [7]: birth, growth, reproduction, and death. As in nature, we intend to execute all these stages in parallel and asynchronously on a population that evolves constantly. The combination of those processes is what we call evolution.

One of the main challenges when designing the algorithm was how to capture the essence of life by reflecting a population evolution?. In our minds, it was clear this task requires a combination of simultaneous working forces that influences the population improvement over time. Our strategy was simple: divide and conquer, looking into the clouds. Enter cloud-computing: previous cloud-computing algorithms have proven successful results working in optimization problems [2].

We turned back to our genetic algorithms knowledge, mixed with our inspiration and cloud computing, and designed the algorithm with the quest to divide the processing workload among multiple computers. This strategy would also make it possible to run our algorithm as a cloud service. Theoretically, one consequence of the workload distribution is to obtain lower execution times.

This algorithm seeks to imitate the natural life cycle, where new individuals are born at any moment and mature over time, where they age and suffer mutations throughout their lives. In reproduction, couples match by mutual attraction, where they may have offspring. Death can happen to everyone: from a newborn to an aged adult, where the individual's fitness will impact their longevity. The general model concept is shown on the Fig. 2.

4.1 Birth

This algorithm starts with a randomly generated population, where the processes will interact with the population independently. This means that at any given time, any individual can experience any of these processes. Birth is the first process of our algorithm, and it is responsible for the initial generation of individuals.

4.2 Growth

As in nature, all individuals constantly grow, mature, or age. With increasing age, individuals may lose strength but also gain more knowledge to solve problems. We represent this with a possible mutation in each increment of age. The growth process will take an individual to assess whether it's ready to mature and undergo changes.

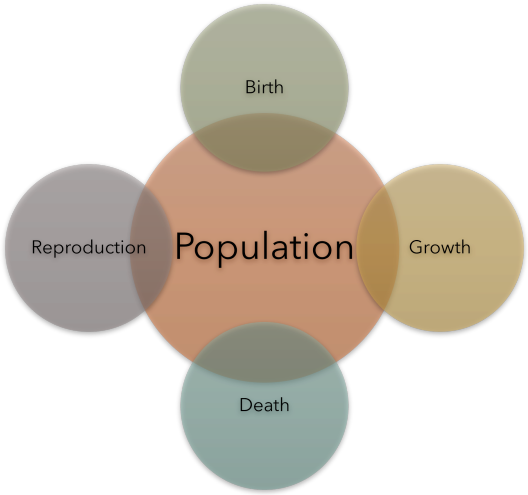

Fig. 2. Algorithm model inspired by the animal species' biological Life-Cycle.

4.3 Reproduction

The attraction of a couple will depend on fitness: the better individual's fitness, the more attractive it will be, making it easier to find mating matches. This process will be taking random pairs of individuals to evaluate their attraction as a couple, to try to breed; when the gestation is successful, a new pair of individuals will be born (as the offspring). Not all couples will be compatible, so reproduction will not always be possible, but the problem arises: how to quantify the attraction between two individuals?

We could have used several strategies to evaluate this attraction. Considering this algorithm takes a similar focus as the study of bacteria growth in microbiology, where we can observe and analyze the evolution of a population over time. As in nature' most species, the bigger a specimen is, the more attractive it is to its mating candidates. To be consistent with both ideas in our algorithm, we use the equation of Newton's Law of Universal Gravitation (1) to calculate the attraction between two individuals, where we visualize the individual fitness as its mass when using the equation. In previous work, Newton's Law of Universal Gravitation has shown success in helping to solve optimization problems.

$$F = G\,\frac{m_1 m_2}{r^2}, \qquad (1)$$

Fig. 3. Newton's Law of Universal Gravitation.

Where:

- **F** is the gravitational force acting between two objects.
- **m1 and m2** are the masses of the objects.
- **r** is the distance between the centers of their masses.
- **G** is the gravitational constant: 6.67430(15)×10-11 m3-kg-1-s-2.

4.4 Death

The death stage represents the challenges and adversities that life presents to overcome. This process evaluates the individual resistance to survive in the environment. The better fitness the individual has will increase its chances of survival. As time progresses, the demands of nature will also increase, pushing for only the best individuals to survive.

4.5 Life-Cycle algorithm flowchart

The Figure 4 describes the general flowchart for the Life-Cycle algorithm.

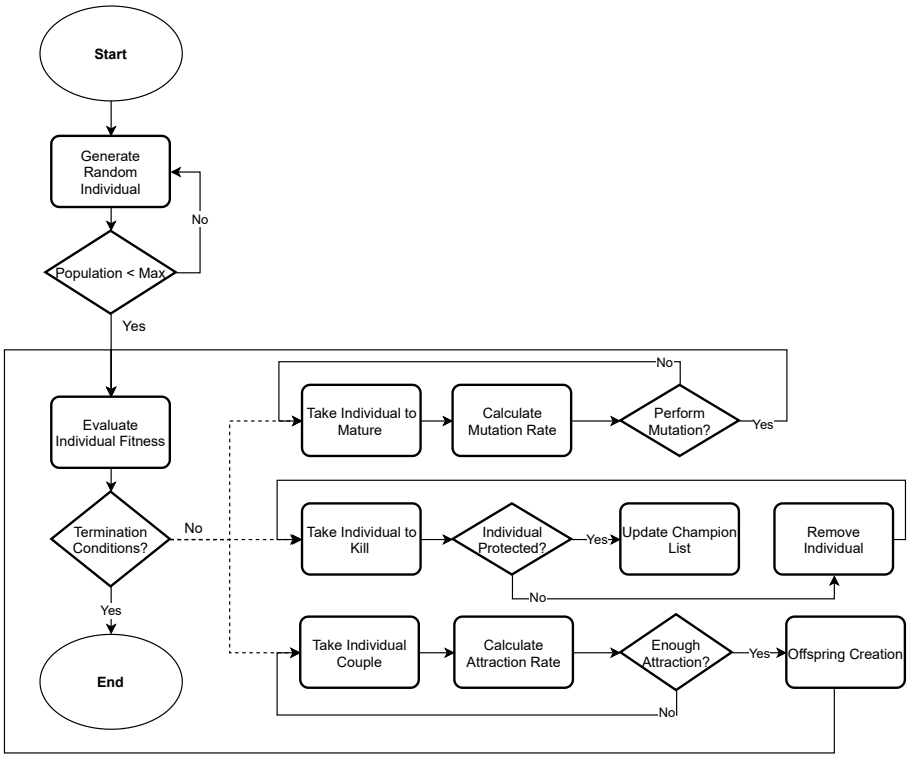

Fig. 4. Life-Cycle algorithm general flowchart.

5 Experiments

As a proof-of-concept, we implemented the algorithm with Docker containers by solving the OneMax problem to compare it with a traditional GA algorithm using the DEAP (Distributed Evolutionary Algorithms in Python) library [1]. The OneMax problem [3, 10] uses a genetic algorithm to evolve a population of randomly generated individuals with zeros and ones, and it stops until a solution of only ones is found.

5.1 Experimental Setup

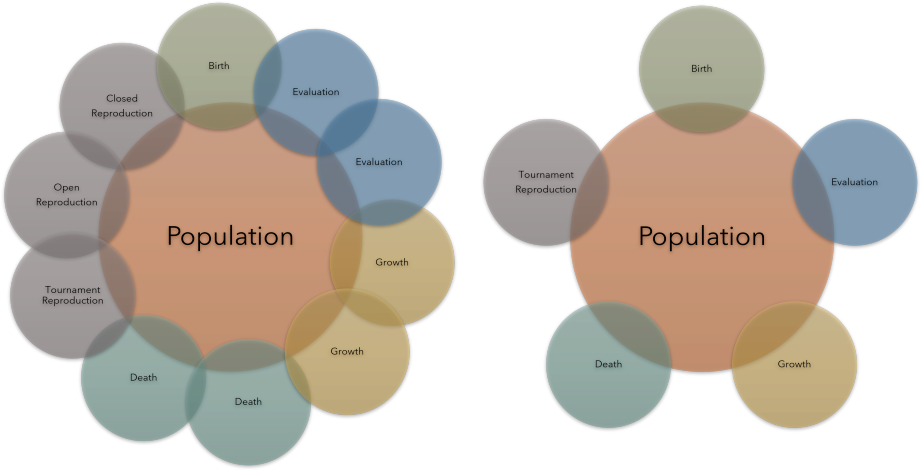

Fig. 5. Life-Cycle algorithm processes in containers comparison.

One strategic difference in our algorithm implementation is the reproduction process, which is flexible to work in parallel with multiple couple selection methods, for example, tournament selection, random selection (Closed Reproduction process), and couple match by creating a new mating individual (Open Reproduction process). We use many combinations of the containerized processes, up to the minimum number required to obtain good results. Figure 5 shows a comparison of only two alternatives for the implementation of the algorithm. Our experimentation started with a configuration of ten processes and ended in five, shown on the left and right sides of Fig. 5, respectively.

5.2 Experiment Configuration

In our four experiments, we needed to match or balance the experimentation parameters according to the specifications of the algorithm used by the DEAP

library. This is to be able to verify if our algorithm would also converge on the solutions in a similar number of evaluations and execution time. Table 1 shows the OneMax initial configuration for DEAP and Life-Cycle experiment.

Table 1. OneMax initial configuration for DEAP and Life-Cycle experiment.

Configuration	**DEAP**	**Life-Cycle**
Population	60	60
Max Generation	20	20
Stagnation	Off	10
Chromosome Length	20	20
Target Fitness	20	20
Crossover Rate	100	100
Mutation Rate	7	7
Tournament Rate	100	50
Tournament Sample	3	4
Open Reproduction Rate	NA	5
Closed Reproduction Rate	NA	45
Max Age	NA	80
Base Approval	NA	80
Goal Approval	NA	100

5.3 Experiment Results

For each experiment, we ran 60 independent executions per algorithm and recorded the following results: Last Generation, Total Evaluations, Time (seconds), Evaluations/second. The labels used on the results of our summarized experiments tables are the following:

- **Last Gen.:** The generation that found the solution.
- **Eval.:** The total number of evaluations the algorithm executed.
- **Time (sec):** Total elapsed or wall clock time, in seconds.
- **Eval/sec:** The calculated rate of evaluations per second.

Experiment 1: The goal of our first OneMax experiment was to make sure our Life-Cycle algorithm worked as expected, converging on the solution. For this experiment, we intentionally used delays (on the tenth of a second magnitude) to follow the Life-Cycle algorithm behavior on the console output. For this experiment, we compared the DEAP algorithm versus the Life-Cycle using ten processes (on containers) where we show the summarized results in Table 2, and Figure 6 shows their Box and Whisker chart.

Experiment 2: The goal of our second experiment was to confirm the Life-Cycle algorithm continued working as expected, converging on the solution. For this experiment, we reduced the time used on delays (now on the thousands

Table 2. Experiment 1 results: OneMax DEAP versus Life-Cycle (10 processes).

Run	OneMax DEAP				Life-Cycle (10P)			
1-60	Last Gen.	Eval.	Time (sec)	Eval/sec	Last Gen.	Eval.	Time (sec)	Eval/sec
Avg.	6.5	391	0.036	10,836	9.6	578	6.830	85

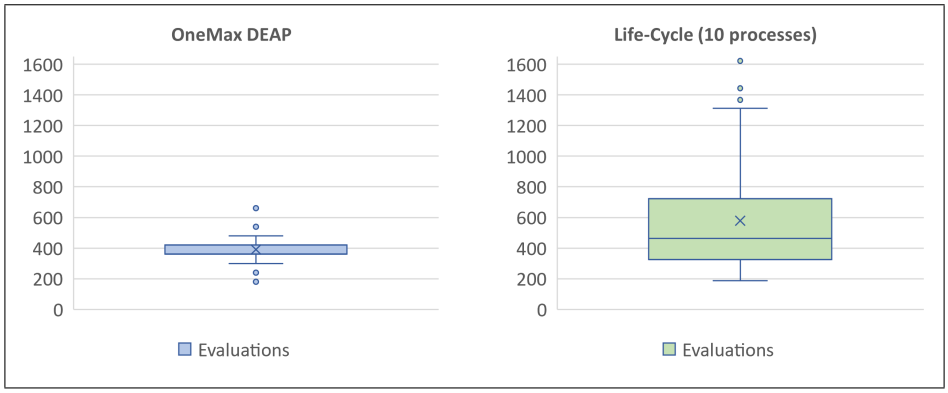

Fig. 6. Box and Whisker chart for the Experiment 1: OneMax DEAP versus Life-Cycle.

of a second magnitude) to follow the Life-Cycle algorithm behavior. For this experiment, we only used tournament selection for the reproduction, on the first configuration running the Life-Cycle on ten processes, versus the second configuration where we reduced the processes to the minimum basic 5. We show the summarized results in Table 3, and Figure 7 shows their Box and Whisker chart.

Table 3. Experiment 2 results, Life-Cycle Tournament selection: 10 versus 5 processes.

Run	LifeCycle (Tourn. 10P)				LifeCycle (Tourn. 5P)			
1-60	Last Gen.	Eval.	Time (sec)	Eval/sec	Last Gen.	Eval.	Time (sec)	Eval/sec
Avg.	7.2	434	0.740	587	8.2	495	0.826	599

Experiment 3: The goal of our third experiment was to follow and study the behavior of the Life-Cycle algorithm that continued converging on the solution. For this experiment, we remained using minimal delays (thousands of a second magnitude). For this experiment, we only used tournament selection for the reproduction, on the first configuration running the Life-Cycle on six processes, two of whom was the Death process, versus the second configuration where we reduced the processes to the minimum (five) but increasing the (Death) goal approval range to 115. We show the summarized results in Table 4, and Figure 8 shows their Box and Whisker chart.

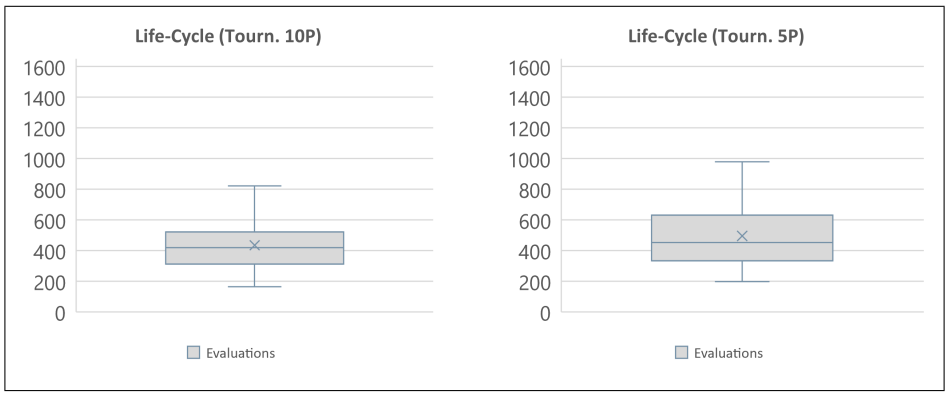

Fig. 7. Box and Whisker chart for the Experiment 2: Life-Cycle Tournament selection (10 versus 5 processes).

Table 4. Experiment 3 results, Life-Cycle Tournament: 6 processes (Death x2) versus 5 processes (80 to 115 goal).

Run	LifeCycle (Tourn. 6P, Death x2)				LifeCycle (Tourn. 5P, 80-115g)			
1-60	Last Gen.	Eval.	Time (sec)	Eval/sec	Last Gen.	Eval.	Time (sec)	Eval/sec
Avg.	8.0	483	0.846	571	6.4	384	0.695	553

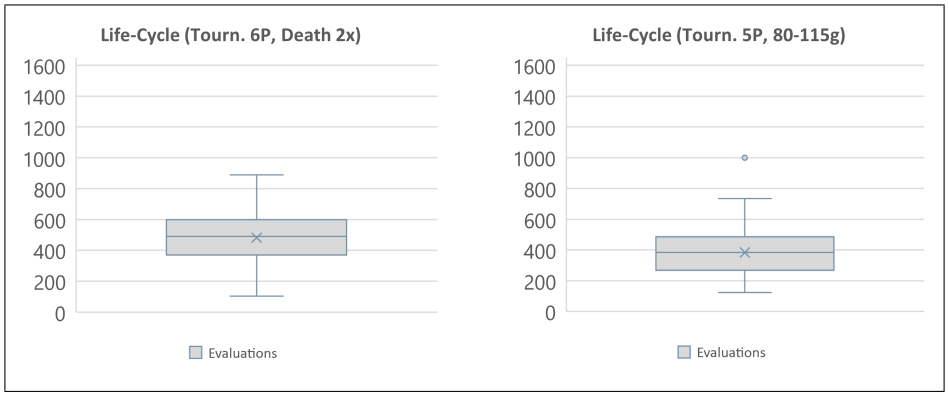

Fig. 8. Box and Whisker chart for the Experiment 3: Life-Cycle Tournament (6 processes double Death, versus 5 processes with an 80 to 115 goal approval).

Experiment 4: The goal of our fourth and last experiment was to follow and study the behavior of the Life-Cycle algorithm that continued converging on the solution. For this experiment, we remained using minimal delays (thousands of a second magnitude). For this experiment, we compared the OneMax DEAP implementation versus the Life-Cycle algorithm, only using tournament selection for the reproduction, with the Life-Cycle configuration running on the minimum (five) processes but increasing the (Death) goal approval range to 125. We show the summarized results in Table 5, and Figure 9 shows their Box and Whisker chart.

Table 5. Experiment 4 results, OneMax DEAP versus Life-Cycle Tournament: 5 processes (80 to 125 goal).

Run	OneMax DEAP				Life-Cycle (Tourn. 5P, 80-125g)			
1-60	Last Gen.	Eval.	Time (sec)	Eval/sec	Last Gen.	Eval.	Time (sec)	Eval/sec
Avg.	6.5	391	0.036	10,836	6.3	377	0.678	556

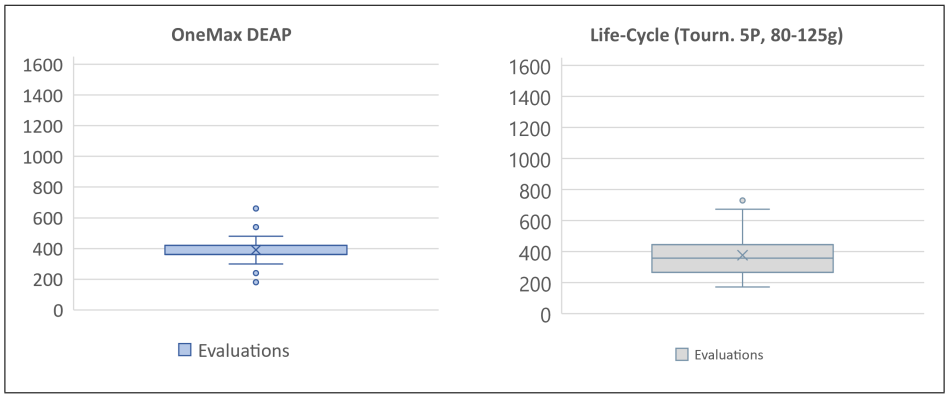

Fig. 9. Box and Whisker chart for the Experiment 4: OneMax DEAP versus Life-Cycle Tournament (5 processes with an 80 to 125 goal approval).

6 Discussion

We must acknowledge the performance time of the DEAP library [1] is highly efficient and short, due in part because the processing work execution is on a single processor with multiple cores, where the communication time is nearly non-existent. In contrast, our algorithm implementation requires a minimum of five containers and a message queue server. Even though the container work has

proven to be very effective and efficient [X], we must consider some time was added to our experiments by the minimal delays (in the thousands of a second magnitude) we used to keep the process execution relatively random, to mimic how the life-cycle stages work in nature [7]. As future work, we could test the Life-Cycle algorithm behavior by eliminating all remaining delays.

In our experiments, we found that when solving simple problems, the cost of communication between containers can become a factor to increase the total performance time, even though, we believe that the opposite must also be true. When solving complex problems, distributing the work on multiple resources should reduce throughput time, making the communication cost negligible.

This scheme allows for multiple parameter's fine tunning, granting us the freedom to experiment with different selection and reproduction strategies simultaneously, which will impact how fast it finds the solution and the obtained quality.

7 Conclusions

We implemented the algorithm with Docker containers by solving the OneMax problem comparing it with a traditional (sequential) GA algorithm, where it showed favorable and promising results. To further validate this work, we could use control or some more complex and demanding problem that requires computing real numbers [8, 6].

As the complexity of problems increases, it is essential to have a scalable, replicable, and fault-tolerant model that uses collaborative techniques to work in the cloud, where multiple resources will be communicating asynchronously. This research has shown that it is possible to evolve a population of individuals, similar to a Genetic Algorithm (GA), using a distributed, parallel, and asynchronous methodology.

References

1. Fortin, F.A., De Rainville, F.M., Gardner, M.A.G., Parizeau, M., Gagné, C.: Deap: Evolutionary algorithms made easy. The Journal of Machine Learning Research **13**(1), 2171–2175 (2012)
2. García-Valdez, M., Merelo, J.J.: Event-driven multi-algorithm optimization: Mixing swarm and evolutionary strategies. In: International Conference on the Applications of Evolutionary Computation (Part of EvoStar). pp. 747–762. Springer (2021)
3. Krejca, M.S., Witt, C.: Lower bounds on the run time of the univariate marginal distribution algorithm on onemax. Theoretical Computer Science **832**, 143–165 (2020)
4. Merelo, J.J., Castillo, P.A., García-Sánchez, P., de las Cuevas, P., Rico, N., García Valdez, M.: Performance for the masses: Experiments with a web based architecture to harness volunteer resources for low cost distributed evolutionary computation. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016. pp. 837–844 (2016)

5. Merelo, J.J., García-Valdez, M., Castillo, P.A., García-Sánchez, P., Cuevas, P., Rico, N.: Nodio, a javascript framework for volunteer-based evolutionary algorithms: first results. arXiv preprint arXiv:1601.01607 (2016)
6. Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., et al.: Evolving deep neural networks. In: Artificial intelligence in the age of neural networks and brain computing, pp. 293–312. Elsevier (2019)
7. Read, K., Ashford, J.: A system of models for the life cycle of a biological organism. Biometrika **55**(1), 211–221 (1968)
8. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary computation **10**(2), 99–127 (2002)
9. Valdez, M.G., Guervós, J.J.M.: A container-based cloud-native architecture for the reproducible execution of multi-population optimization algorithms. Future Generation Computer Systems **116**, 234–252 (2021)
10. Witt, C.: Upper bounds on the running time of the univariate marginal distribution algorithm on onemax. Algorithmica **81**(2), 632–667 (2019)