Applied Artificial Intelligence and Robotics Using Revolve Framework in R and Python

Daniel Zito, Vadim Mazitov, Gayathri Adhimoolam, Sunitha Radhakrishnan

Master’s in Automation & IT

Cologne University of Applied Sciences, Germany
sig.danielzito@gmail.com, vadmazitov@gmail.com,
gayuathi97@gmail.com, sunitha0594@gmail.com

Abstract: Evolutionary Algorithms base their approach to imitate most of the aspects of natural evolution. EAs are considered sets of heuristics, self-learning techniques to improve performance. In this paper, an investigation of a practical EA regarding modular robots was made, and several experiments using different robot morphologies were conducted. To simulate and evaluate the performance of the robots Revolve/Gazebo framework was used. The performance of each robot was measured with respect to its fitness, a number value indicating the speed at which the robot can move. The goal of this work was to set up an interface between Docker and R, employ an EA and improve the algorithm steps based on the results of the experiments. To speed up the evaluation step of the robots, these were evaluated in batch, using multiple PC cores. The performances of different types of robots ware evaluated and the results were analyzed. Experiments were run by considering several parameters such as the budget available, the mutation step size as well as the order of the algorithm steps. Finally, the paper shows how limiting the mutation step according to a given threshold and simply changing the order of the algorithm steps results in better robot performances.

Keywords: Evolutionary Algorithm, Modular Robots, Revolve/Gazebo, Fitness.

1 Introduction

Evolutionary computing is undergoing a rapidly growing success in the Artificial Intelligence world. Indeed, Evolutionary Algorithms have become a promising next phase towards a revolutionary, innovative type of AI that can make autonomous decisions, adapt to a changing environment, and find non-obvious solutions to complex situations [1]. Evolutionary robotics involves the selection, variation and inheritance concepts of natural evolution to the structure of robots with embodied intellectual ability. It can be considered as a sub-field of robotics aimed at creating more resilient and adaptive robots [2]. The main idea of evolutionary robotics is to generate a group of individuals with various genomes, each specifying the features of the robot's control system or its structure [3].

Several different aspects must be addressed concurrently in the creation of a robot: its morphology, sensory equipment, motor mechanism, control architecture, etc. The work illustrated in this paper primarily focuses on the aspect regarding the morphology of the robot. Indeed, the structure of the robot is directly connected to its performance.
In this paper, performance is intended for the fitness of the robot, a value indicating how fast the robot can move. For this scope, an evolutionary algorithm was built, and several simulation experiments were conducted based on different robot morphologies. The Evolutionary Algorithm was implemented to optimize the performance of the robots by changing their modular structure. A remodeled version of the same EA was additionally implemented. The idea was to explore different combinations of parameters and understand how they affect the performance of the algorithm in terms of final output and computational time. Besides the general EA parameters, such as mutation step size, and budget size, the revised algorithm takes into account the time required for producing an output. To speed up the mutation and evaluation steps, the algorithm relies on all the PC cores available, excluding the core needed for the PC to work. Indeed, using more cores allows the algorithm to recombine, mutate, and evaluate the population of robots in batches, resulting in speed gains. Ultimately, the results of both algorithms were analyzed and compare in terms of performance.

2 Methodology and Implementation

2.1 Evolutionary Algorithm Overview

In Evolutionary Algorithms, every individual denotes the search point in the region of an effective solution to the given problem. The maximum of this evolutionary field follows the approach of the classical algorithms. However, there are different ways of running experiments depending on the final goal of the application. The existing EA was modified to improve the performance of the robot considering fitness value. Although these options differ in numerous aspects, in any case, the methodology would follow the same general principles of experimenting and testing. Evolutionary Algorithms are stochastic by nature. Therefore, experiments were executed multiple times, under the same conditions, by only varying few parameters at the time to get the most desirable fitness value of the robot.
To analyze and evaluate the performance of each individual, a fitness value was assigned. The probability of an individual to be selected as a parent depends on its fitness value. Individuals with higher fitness values have more probability to be chosen. This approach increases the probability that the offspring generated will obtained a better modular structure by the preceding generation. A real-world Evolutionary Algorithm was addressed to optimize the morphology of modular robots. The EA structure is depicted in **Fig. 1**.

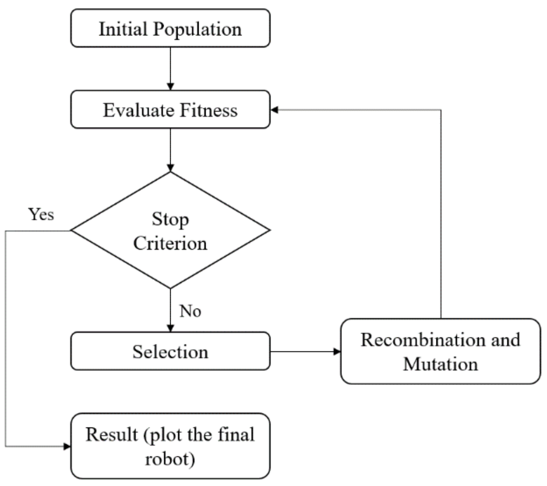


Fig. 1. General schema of an EA. Recombination and Mutation steps may be alternated

The population is initialized by generating random robots. The structure of the robots consists of different types of blocks, each assigned with a number. The number 1 designates the *core* of the robot, the number 2 a *movable block*, and the number 3 a *docking block*. The core part of the robot is non-modifiable. Besides, there are four possible orientations and four possible dock points to the core. Then each individual is evaluated, its fitness value measured, and then parent selection takes place.
In the Mutation step, either a block (3) or a movable part (2) is added or removed randomly to one of the dock points, according to the orientation. Recombination (or crossover) means combining two individuals to form a third-new individual. Finally, the individuals are ordered and sorted based on their fitness value. The robot with the highest fitness value goes on top and the others follow. The selection of survival aims to select a subset of well-performing individuals from the overall population set. The best robot is then plotted. Hence, the algorithm ends, and a new robot that possibly exhibits better performance is created. Then the cycle repeats for multiple experiments.

2.2 Software Involved and Overall Implementation Structure

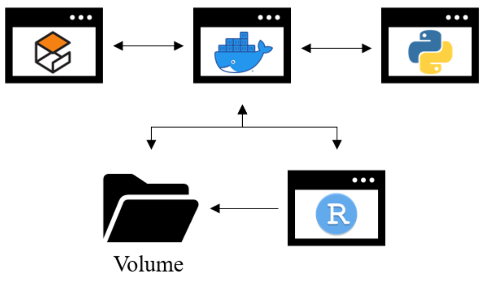


Fig. 2. Overall implementation structure. Software used (from left to right): Gazebo simulator, Docker, Python, R. The Volume folder works as a bridge between Docker and R

Fig. 2. represents the overall structure of the project. RStudio, an Integrated Development Environment, was represented the core of the entire work. The Evolutionary Algorithm, to generate, evaluate, recombine, and mutate robots, was programmed using the R programming language. The initial population was generated randomly to produce all the possible forms of robots. A shared volume folder was created in the local machine to store the robots generated during all the different stages of the evolution process. Whenever the algorithm fails during the evolution process, the robots generated so far at failure point are still found in the folder. Hence, the robots can be passed as the initial population for the succeeding run, thereby reducing computational time.
To simulate and to test the performance of the robots generated, the Gazebo simulator was utilized. As the installation of the Gazebo simulator in a Windows environment is a tedious process, Docker, a Linux based environment, was used to manage the Gazebo simulator running into a virtual machine. Gazebo was built as a container in Docker and an image was created for every instance. An interface between RStudio and the Docker terminal was designed by invoking the system command in R. A Docker file with an environment variable describing the type of robot to evaluate was created. The shared volume folder created was linked to the Docker terminal to transfer robots for the evaluation of their fitness value.
The performance of each robot was analyzed through the fitness value, which provides information about how fast a robot can move. To estimate the fitness of each robot, preprogrammed Python scripts were used. Each robot was given a lifetime of 30 seconds to prove its efficiency inside the simulation environment provided by Gazebo.
Hence, as robot generated in RStudio was stored in .yaml format into the shared volume folder. Through the interface running between Docker and RStudio, the Docker terminal fetched the stored robot to be evaluated inside the simulation environment. After the simulation phase, the fitness value of each robot was extracted from the output lines retrieved by the system command from the Docker terminal. The robots were ordered according to the fitness value and the best robot generated was plotted.
To increase the efficiency of the algorithm in terms of computation time, batches of robots were sent for parallel evaluation. This was accomplished by using the maximum number of cores of the PC. It must be mentioned that it is necessary to leave one core free for internal PC computation.

2.3 Evolutionary Algorithm Implementation

Initial implementation

Originally, the algorithm was able to process only one robot at the time using a single core of a cluster to execute it. The EA interface was common and not application-specific. The modified algorithm was thought to process robots so that calculations could be distributed over the whole cluster to accelerate the execution by number-of-core times.

Concurrency integration

No real-life application can be designed without the ability to work in a concurrent environment to efficiently use the computation power provided. This task turns out to be quite challenging in the R environment, as it is a one-threaded programming language and does not provide native tools to implement multi-threading.
To fulfill a concurrency implementation, the R library *doParallel* was employed. Since it is not feasible to generate more than one executing thread, the library generates different copies of R and executes them in parallel. To adapt the algorithm to a new concurrent environment, the interface of general functionalities (population initialization, evaluation, recombination, and mutation) needed to be changed to support multiple robots. The pattern followed is known in Software Engineering as *Adapter.* The idea is to wrap the existing functions and apply additional logic to them. Besides, a naming pattern was used to highlight functions that can accept and return a list of robots, names of such functions end with the word *-Batch.*

Parameter explanation

For the implementation of the algorithm, the following parameters were considered:

- *fun*: represents the objective function used to evaluate the robot fitness
- *budget*: represents the total number of evaluations available
- *plotBestAfter*: determines whether a plot of the best robot will be shown at the end of every iteration or after the budget is exhausted
- *popSize*: represents the population size
- *nCores*: represents the number of robots to execute at a time.
- *removedRate*: represents the part of a robot (in %) that can be removed
- *numberOfTrials*: related to removedRate, represents the number of trials removedRate can be executed before aborting it (this happens in case of invalid robots generated)
- *oldPop*: represents the initial population (if set to NULL, the population will be randomly generated). It is useful to start with an already-existing population in case the previous algorithm execution brakes
- *return*: represent the best-generated robot of the best-generated population in case an error occurs during the execution

Initialization

The Initialization phase is when the robots are generated and it can be performed in two ways:

- *oldPop* is not passed: random robots are created equal in number to *popSize* specified
- *oldPop* is passed: oldPop list of robots is used as the initial population. Random robots are also added to reach popSize, if necessary

Once all the new robots are generated, they will be evaluated in parallel, according to the batch size. The new robots, along with their corresponding fitness value, will be added to the population list.

Recombination

In the original implementation of the algorithm, the Recombination step was performed after the Mutation step. However, significant improvements were observed after placing Recombination before Mutation.
For the Recombination, 2**nCores* robots are chosen. The probability of a robot being chosen is based on its fitness value, the higher its fitness value the higher the probability. These robots are then recombined, evaluated in parallel, and added to the population list. Finally, the robots will be sorted according to their fitness values.

Mutation Addition

In the Mutation Addition step, a new block or a movable part is randomly added to the robot chosen in the Recombination step and added to the population list. Then robots are sorted according to their fitness values.

Mutation Removal

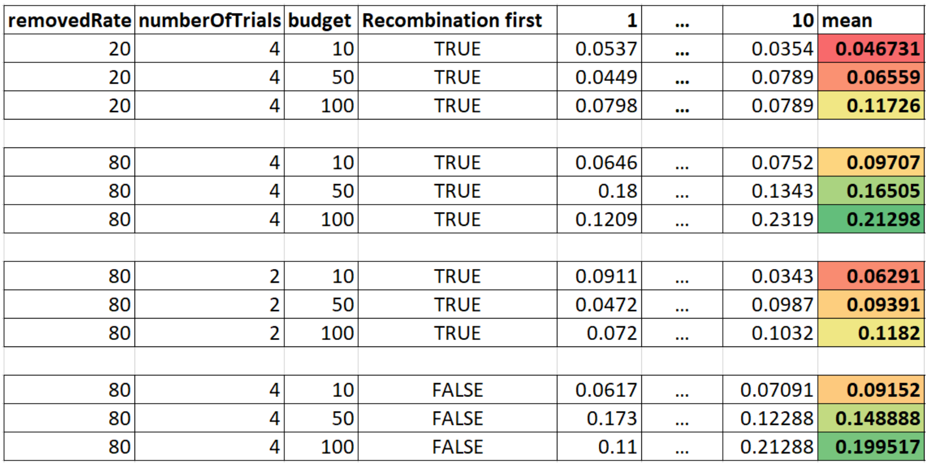
In the Mutational Removal step, a block or a movable part of the robot chosen in the Recombination step is randomly removed. This might lead to the removal of other parts attached to the specific block or producing an invalid robot. To limit this action, two parameters were introduced. *removedRate* and *numberOfTrials (*refer to Parameter explanation*).* Finally, the mutated robots are evaluated and added to the population list.

Population Selection

After sorting the population, the first *popSize*-number of robots is selected to take part in the next iteration.
The whole function is wrapped inside a try-catch block to save the last generated population in case the algorithm fails so that all the robots processed by the algorithm are not lost.

3 Experiments and Results

For evaluating the performance of the original algorithm, several experiments were performed by varying *budget*, the order of the algorithm steps, *removedRate*, and *numberOfTrials* parameters. The results were analyzed comparing the fitness value of the best robot generated after each experiment.
The original algorithm was compared to the revised algorithm to examine the improvements in terms of the robot's performance. For each combination of parameters, numerous runs were executed for statistical significance.
To understand and interpret the outcomes of the experiments conducted, different boxplots were analyzed and compared. The experiments are grouped in different cases.



removedRate	numberOfTrials	budget	Recombination first	1	...	10	mean
20	4	10	TRUE	0.0537	...	0.0354	0.046731
20	4	50	TRUE	0.0449	...	0.0789	0.06559
20	4	100	TRUE	0.0798	...	0.0789	0.11726
80	4	10	TRUE	0.0646	...	0.0752	0.09707
80	4	50	TRUE	0.18	...	0.1343	0.16505
80	4	100	TRUE	0.1209	...	0.2319	0.21298
80	2	10	TRUE	0.0911	...	0.0343	0.06291
80	2	50	TRUE	0.0472	...	0.0987	0.09391
80	2	100	TRUE	0.072	...	0.1032	0.1182
80	4	10	FALSE	0.0617	...	0.07091	0.09152
80	4	50	FALSE	0.173	...	0.12288	0.148888
80	4	100	FALSE	0.11	...	0.21288	0.199517

Fig. 4. Overview containing all the experiments conducted. For each set of parameters, 10 experiments are repeated. The color indicates the fitness index, from red (low fitness value) to green (high fitness value)

Case 1: Change in *removedRate* and *budget* Parameters

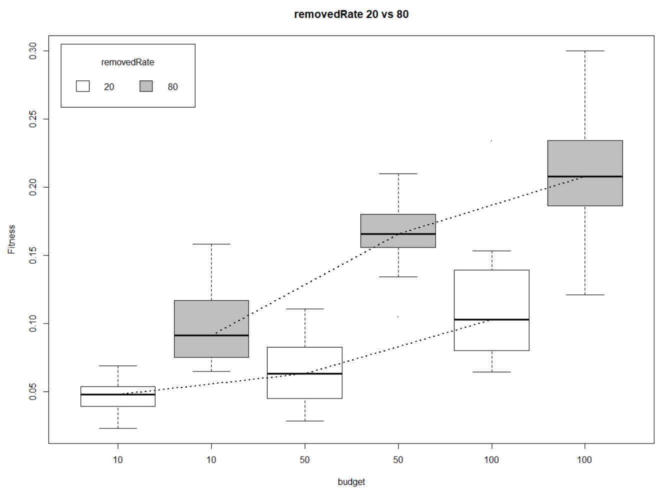


Fig. 5. Boxplots comparing changes in *removedRate* and *budget*. Grey boxes are for *removedRate* 80 while white boxes are for *removedRate* 20. Dotted trendlines pass through the median value of each box

Fig. 5 shows how increasing the value of *removedRate* resulted in higher fitness values. Indeed, in each of the three different cases, the grey boxplots are located higher than the white boxes. Moreover, the trendlines raise as *budget* increases for both *removedRate* 20 and *removedRate* 80.

Case 2: Change in *numberOfTrials*

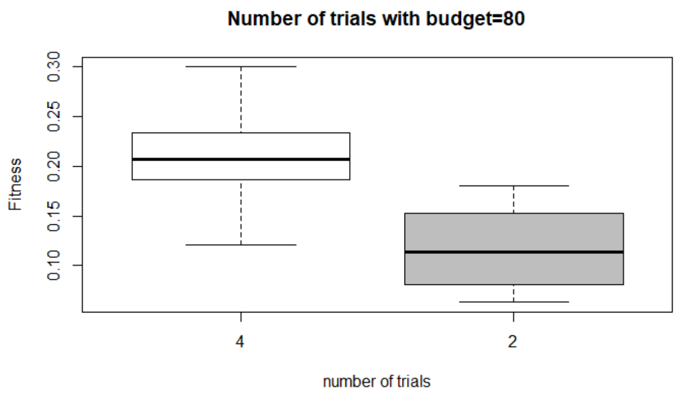


Fig. 6. Boxplots comparing changes in *numberOfTrials*, 2 vs 4.

Fig. 6 shows how reducing *numberOfTrials* resulted in lower fitness values. *numberOfTrials* indicates the number of times the Mutation Removal step is attempted. The higher this value the more the probability the Mutation Removal step actually occurs. The Mutation step can be not successful, in case an invalid robot is generated, for example. This may occur in both the Mutation Removal and the Mutation Addition step. *numberOfTrials* is related to *removedRate* as both the parameters take into account removing parts from the robots

Case 3: Swap Mutation and Recombination Step

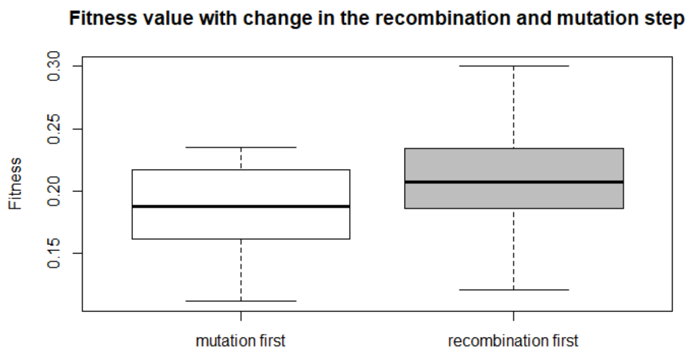


Fig. 7. Boxplots comparing the order of the algorithm steps. In the revised algorithm the Recombination step is placed before the Mutation step

The boxplots, in **Fig. 7**, show how swapping Recombination and Mutation step turned out to slightly increase the performance of the robot. In the original algorithm, the robots were mutated and then recombined, while in the revised algorithm the offspring is mutated.

Case 4: Comparing The Computation Time Based on The Number of Cores

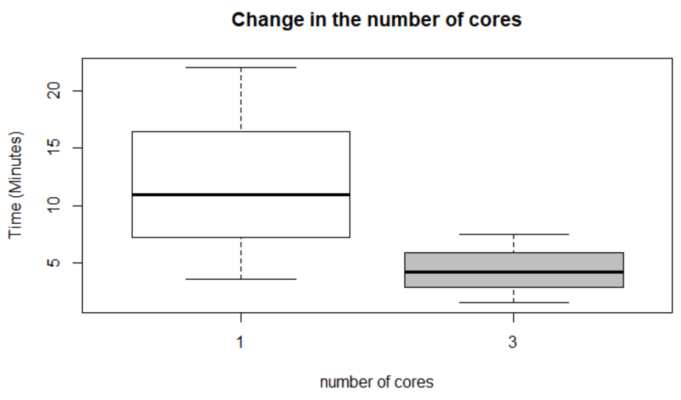


Fig. 8. Boxplots comparing the time (in mins) taken by the algorithm to run. In the case shown *budget* was set to 50

Increasing the number of cores utilized to run the algorithm allowed us to process (generate, evaluate, recombine, and mutate) multiple robots at the time. The idea was to process robots in batches. At a time *nCores* were processed. By comparing the case with 3 cores used with the case with only 1 core used, it can be observed that a great amount of time is saved. Indeed, roughly only one-third of the computation time is required when using 3 cores, compared to the computation time when using 1 core.

4 Conclusion and Future Work

In this research, the implementation and analysis of the Evolutionary Algorithm for modular robots were made. The conclusions from the experiments are:

- An increase in the budget increases the fitness of the robot.
- An increase in the removed rate in the Mutation Removal step results in better fitness value.
- An increase in the number of trials for the mutation removal also increases the fitness of the robots.
- Increasing the number of cores of the PC reduces the time taken by the evolutionary algorithm to generate and evaluate the robots due to the batch evaluation of robots during each step.
- Performing recombination before mutation in the EA also has a positive impact on the performance of the robots.

- Implementing the try-catch block helps in saving the last generated population of robots in the folder whenever the algorithm fails.

The scope of the work in the future would be to analyze the relationship between the descriptors and the fitness value of the robots. Besides, starting with existing well-performing robots as the initial population would improve the performance of the next generation of robots resulting in even better final robots. Moreover, using the Sequential Parameter Optimization Toolbox (SPOT) to optimize the parameters such as *removedRate*, *popSize* and, *numberOfTrials* might increase the performance of the algorithm.

5 References

1. Evolving Neural Networks through Augmenting Topologies. Kenneth O. Stanley and Risto Miikkulainen. Evolutionary Computation. 2002 10:2, 99-127
2. Evolutionary Robotics: What, Why, and Where to. Doncieux Stephane, Bredeche Nicolas, Mouret Jean-Baptiste, Eiben Agoston E. (Gusz). Vol. 2. 2015. p. 4.
3. Floreano D, Keller L. Evolution of adaptive behaviour in robots by means of Darwinian selection. PLoS Biol. 2010;8(1): e1000292. Published 2010 Jan 26. doi: 10.1371/journal.pbio.1000292
4. D'Angelo M., Weel B., Eiben A.E. (2013) Online Gait Learning for Modular Robots with Arbitrary Shapes and Sizes.In: Dediu AH.,Martin-Vide C.,Truthe B.,Vega-Rodriguez M.A (eds) Theory and Practice of Natural Computing. TPNC 2013, Lecture Notes in Computer Science, vol 8273. Springer, Berlin, Heidelberg.
5. Zbigniew Michalewicz, Chapter 8: How to solve it – Evolutionary Algorithms and Optimization, Published on 2004