DEAP - Enabling Nimbler Evolutions

François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné
Département de génie électrique et de génie informatique - Université Laval - Québec (Québec), Canada

DEAP is a Distributed Evolutionary Algorithm (EA) framework written in Python and designed to help researchers developing custom evolutionary algorithms. Its design philosophy promotes explicit algorithms and transparent data structures, in contrast with most other evolutionary computation softwares that tend to encapsulate standardized algorithms using the black-box approach. This philosophy sets it apart as a rapid prototyping framework for testing of new ideas in EA research. An executable notebook version of this paper is available at `https://github.com/DEAP/notebooks`.

Highlights

- Building blocks for testing ideas
- Rapid prototyping
- Fully transparent
- Parallel ready
- Exhaustively documented
- Available at http://deap.gel.ulaval.ca

Introduction

The DEAP framework [1; 2] is designed over the three following founding principles:

1. Data structures are key to evolutionary computation. They must facilitate the implementation of algorithms and be easy to customize.
2. Operator selection and algorithm parameters have strong influences on evolutions, while often being problem dependent. Users should be able to parametrize every aspect of the algorithms with minimal complexity.
3. EAs are usually embarrassingly parallel. Therefore, mechanisms that implement distribution paradigms should be trivial to use.

With the help of its sister project SCOOP [3] and the power of the Python programming language, DEAP implements these three principles in a simple and elegant design.

Data Structures

A very important part of the success for designing any algorithm — if not the most important — is choosing the appropriate data structures. Freedom in type creation is fundamental in the process of designing evolutionary algorithms that solve real world problems. DEAP's *creator* module allows users to:

- create classes with a single line of code (inheritance);
- add attributes (composition);
- group classes in a single module (sandboxing).

In the following listing, we create a minimizing fitness.

```
from deap import base, creator
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

The `create` function expects at least two arguments; the name of the class to be created and the base class it inherits from. The next arguments are used as class attributes. Thus, the class just created is a `FitnessMin` inheriting from the base class `Fitness` and having a `weights` attribute set to the one element tuple $(-1.0,)$, indicating minimization of a single objective. The trailing comma is mandatory to define a single item tuple in Python. A multi-objective fitness would be created using a multi-element tuple.

Next, we define with the same mechanism an `Individual` class inheriting from a `list` and composed with a `fitness` attribute.

```
creator.create("Individual", list, fitness=creator.FitnessMin)
```

When an `Individual` is instantiated, its `fitness` is initialized as an instance of the previously defined `FitnessMin` class. This is illustrated in the following example,

```
ind = creator.Individual([1,0,1,0,1])
ind.fitness.values = (sum(ind),)
```

where an individual is created from a list of binary values and the value of its fitness is set to the sum of its elements. In DEAP, the fitness value is always multi-objective with the single objective case being a tuple of one element.

Operators

Operator selection is another crucial part of evolutionary algorithms. It must be straightforward and its parametrization intuitive. DEAP's *Toolbox* enables users to:

- create aliases for operators;
- register operators' parameters;
- interchange operators efficiently;
- regroup all operators in a single structure.

The next example presents the construction of a toolbox and how operators and their parameters are registered.

```
from deap import tools
toolbox = base.Toolbox()
toolbox.register("mate", tools.cxOnePoint)
toolbox.register("mutate", tools.mutGaussian, mu=0.0, std=1.0)
```

The `register` function expects at least two arguments; the alias of the function and the function itself. The next arguments are passed to the function when called, similarly to the `partial` function from the standard *functools* module. Thus, the first operator is a one point crossover registered under the alias `mate`. The second operator, a gaussian mutation, is registered with its parameters under the generic name `mutate`. Both operators are available from the *tools* module along with many more instruments supporting evolution that are presented at the end of this paper.

During subsequent experiments, replacing the one point crossover by a two point crossover is as easy as substituting the third line of the previous listing by the following one.

```
toolbox.register("mate", tools.cxTwoPoint)
```

Wherever the generic function `mate` is used, the new two point crossover will be used.

Parallelization

DEAP is parallel ready. The idea is to use a mapping operation that applies a function to every item of a sequence, for instance to evaluate the fitnesses. By default, every toolbox is registered with the standard `map` function of Python. For algorithms to evaluate individuals in parallel, one only needs to replace this alias by a parallel map such as the one provided by SCOOP [3], a library capable of distributing concurrent tasks on various environments, from grids of workstations to supercomputers.

```
from scoop import futures
toolbox.register("map", futures.map)
```

DEAP is also compatible with the standard *multiprocessing* module, if the user only cares to run on a single computing node with multiple cores.

```
import multiprocessing
pool = multiprocessing.Pool()
toolbox.register("map", pool.map)
```

With these powerful tools, DEAP allows scientists and researchers with little programming knowledge to easily implement distributed and parallel EAs.

Preaching by Example

The best introduction to evolutionary computation with DEAP is to present simple, yet compelling examples. The following sections set forth how algorithms are easy to implement while keeping a strong grip on how they behave. The first section introduces a classical genetic algorithm and exposes different levels of explicitness. The second section presents how genetic programming is implemented in DEAP and the versatility of the GP module. The final example demonstrates how easy it is to implement a generic distributed island model with SCOOP.

A Simple Genetic Algorithm

A commonly used example in evolutionary computation is the OneMax problem which consists in maximizing the number of ones in a binary sequence. The more ones an individual contains, the higher its fitness value is. Using a genetic algorithm to find such an individual is relatively straightforward. Applying crossovers and mutations on a population of randomly generated binary individuals and selecting the fittest ones at each generation usually converge to a perfect (all ones) solution. A problem of this simplicity should be solved with a very simple program.

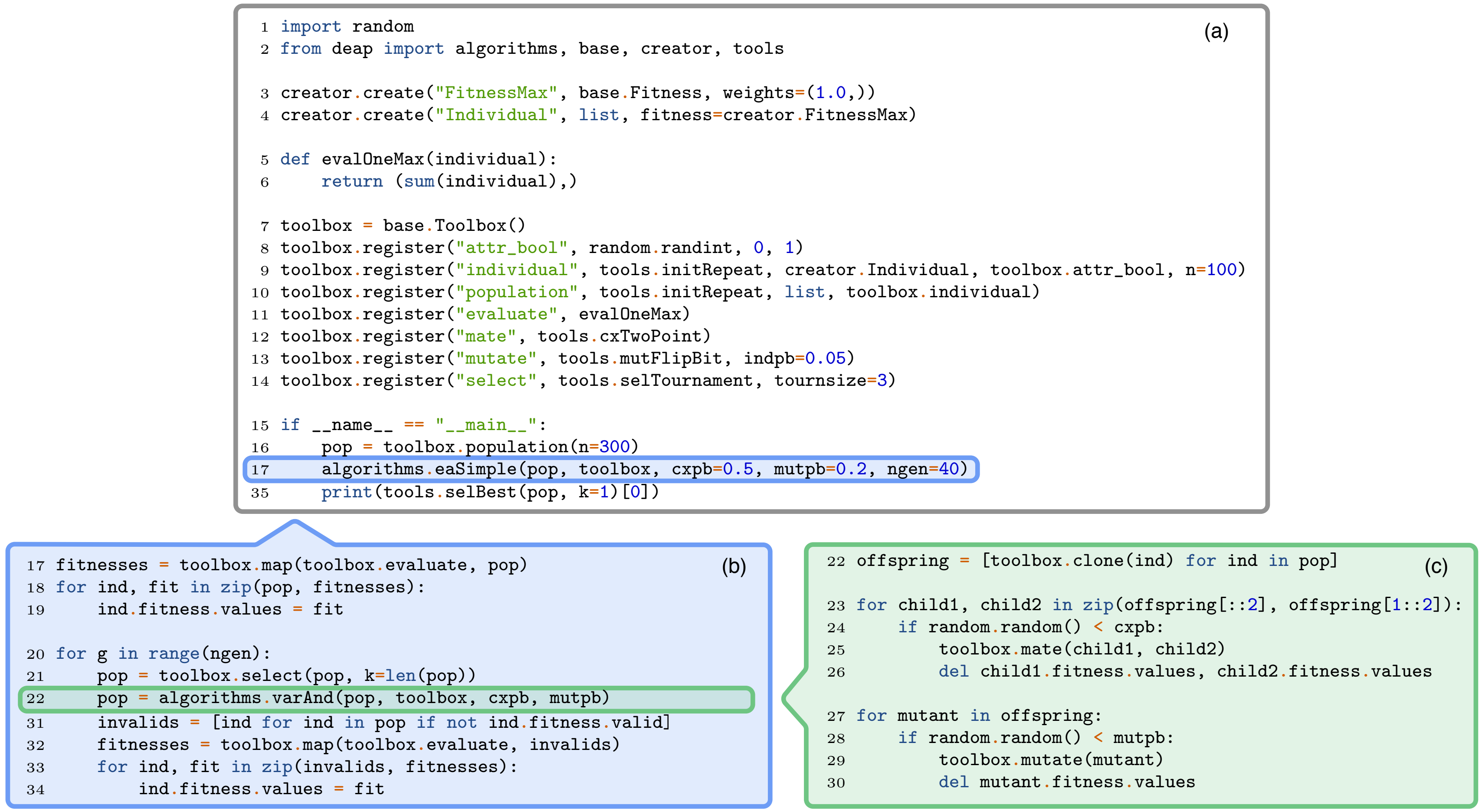
Figure 1(a) presents all that is needed to solve the OneMax problem with DEAP. The first two lines import the necessary modules. Next, on lines 3 and 4, two types are created; a maximizing fitness (note the positive weights), and a list individual composed with an instance of this maximizing fitness. Then, on lines 5 and 6, the evaluation function is defined. It counts the number of ones in a binary list by summing its elements (note again the one element returned tuple corresponding to a single objective fitness). Subsequently, a `Toolbox` is instantiated in which the necessary operators are registered. The first operator, on line 8, produces binary values, in this case integers in $[0,1]$, using the standard *random* module. The alias `individual`, on line 9, is assigned to the helper function `initRepeat`, which takes a container as the first argument, a function that generates content as the second argument, and the number of repetitions as the last argument. Thus, calling the individual function instantiates an `Individual` of `n=100` bits by calling repeatedly the registered `attr_bool` function. The same repetition initializer is used on the next line to produce a population as a list of individuals. The missing number of repetitions `n` will be given later in the program. Subsequently, on lines 11 to 14, the evaluation, crossover, mutation and selection operators are registered with all of their parameters.

The main program starts at line 16. First, a population of `n=300` individuals is instantiated. Then, the algorithm, provided with the population and the toolbox, is run for `ngen=40` generations with `cxpb=0.5` probability of mating and `mutpb=0.2` probability of mutating an individual for each generation. Finally, on line 35, the best individual of the resulting population is selected and displayed on screen.

Controlling Everything

When developing, researching or using EAs, pre-implemented canned algorithms seldom do everything that is needed. Usually, developers/researchers/users have to dig into the framework to tune, add or replace a part of the original algorithm. DEAP breaks with the traditional black-box approach on that precise point; it encourages users to rapidly build their own algorithms. With the different tools provided by DEAP, it is possible to design a nimble algorithm that tackles most problems at hand.

Starting from the previous OneMax solution of Figure 1(a), a first decomposition of the algorithm replaces the canned `eaSimple` function (line 17) by the generational loop illustrated in Figure 1(b). Again, this example is exhaustive but still very simple. On the first 3 lines, the evaluation function is applied to every individual in the population by the `map` function contained in every toolbox. Next, on line 18, a loop over both the population and the evaluated fitnesses sets each individual's fitness value. Thereafter, on line 20, the generational loop begins. It starts by selecting `k` individuals from the population. Then, the selected individuals are varied by crossover **and** mutation using the `varAnd` function. A second variation scheme `varOr` can also be used, where the individuals are produced by crossover **or** mutation. Once modified, the individuals are evaluated for the next iteration. Only freshly produced individuals have to be evaluated; they are filtered by their fitness validity; `valid` property of the fitness (line 31). This version of the program provides the possibility to change the stopping criterion and add components to the evolution.



```python
1  import random
2  from deap import algorithms, base, creator, tools

3  creator.create("FitnessMax", base.Fitness, weights=(1.0,))
4  creator.create("Individual", list, fitness=creator.FitnessMax)

5  def evalOneMax(individual):
6      return (sum(individual),)

7  toolbox = base.Toolbox()
8  toolbox.register("attr_bool", random.randint, 0, 1)
9  toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, n=100)
10 toolbox.register("population", tools.initRepeat, list, toolbox.individual)
11 toolbox.register("evaluate", evalOneMax)
12 toolbox.register("mate", tools.cxTwoPoint)
13 toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
14 toolbox.register("select", tools.selTournament, tournsize=3)

15 if __name__ == "__main__":
16     pop = toolbox.population(n=300)
17     algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40)
35     print(tools.selBest(pop, k=1)[0])
```

```python
17 fitnesses = toolbox.map(toolbox.evaluate, pop)
18 for ind, fit in zip(pop, fitnesses):
19     ind.fitness.values = fit

20 for g in range(ngen):
21     pop = toolbox.select(pop, k=len(pop))
22     pop = algorithms.varAnd(pop, toolbox, cxpb, mutpb)
31     invalids = [ind for ind in pop if not ind.fitness.valid]
32     fitnesses = toolbox.map(toolbox.evaluate, invalids)
33     for ind, fit in zip(invalids, fitnesses):
34         ind.fitness.values = fit
```

```python
22 offspring = [toolbox.clone(ind) for ind in pop]

23 for child1, child2 in zip(offspring[::2], offspring[1::2]):
24     if random.random() < cxpb:
25         toolbox.mate(child1, child2)
26         del child1.fitness.values, child2.fitness.values

27 for mutant in offspring:
28     if random.random() < mutpb:
29         toolbox.mutate(mutant)
30         del mutant.fitness.values
```

Fig. 1: OneMax example with DEAP. (a) Simpler version relying on the pre-implemented eaSimple algorithm. (b) Unboxing of the eaSimple algorithm to control selection, variation and evaluation. (c) Unfolding of the variation to handle crossover and mutation.

An even greater level of detail can be obtained by substituting the `varAnd` function by its full content, presented in Figure 1(c). This listing starts with the duplication of the population by the `clone` tool available in every toolbox. Then, the crossover is applied to a portion of consecutive individuals. Each modified individual sees its fitness invalidated by the deletion of its values on line 26. Finally, a percentage of the population is mutated and their fitness invalidated. This variant of the algorithm provides control over the application order and the number of operators, among other aspects.

The explicitness in which algorithms are written with DEAP clarifies the experiments. This eliminates any ambiguity on the different aspects of the algorithm that could, when overlooked, jeopardize the reproducibility and interpretation of results.

Genetic Programming

DEAP also includes every component necessary to design genetic programming algorithms with the same ease as for genetic algorithms. For example, the most commonly used tree individual can be created as follows:

```
import math, operator
from deap import base, creator, tools, gp

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree,
     fitness=creator.FitnessMin)
```

The primitive tree is provided in the *gp* module since it is one of the few data types the Python standard library does not provide. The primitives and terminals that will populate the trees are regrouped in a primitive set. The following listing presents a primitive set instantiation with basic operators provided by the standard library *operator* module. The `arity` of a primitive is its number of operands.

```
pset = gp.PrimitiveSet(name="MAIN", arity=1)
pset.addPrimitive(operator.add, arity=2)
pset.addPrimitive(operator.sub, arity=2)
pset.addPrimitive(operator.mul, arity=2)
pset.addPrimitive(operator.neg, arity=1)
```

Functions that initialize individuals and populations are registered in a toolbox just as in the preceding genetic algorithm example. DEAP implements the three initialization methods proposed by Koza [4] to generate trees: full, grow, and half-and-half.

```
toolbox = base.Toolbox()
toolbox.register("expr", gp.genFull, pset=pset, min_=1, max_=3)
toolbox.register("individual", tools.initIterate,
        creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat,
        list, toolbox.individual)
```

We may now introduce an example of a symbolic regression evaluation function.

```
def evaluateRegression(individual, points, pset):
    func = gp.compile(expr=individual, pset=pset)
    sqerrors = ((func(x) - (x**4 + x**3 + x**2 + x))**2 for x in points)
    return (math.sqrt(sum(sqerrors) / len(points)),)
```

First, the `gp.compile` function transforms the primitive tree into its executable form, a Python function, using a primitive set `pset` given as the evaluation function's third argument. Then, the rest is simple math: we compute the root mean squared error between the individual's program and the target $x^4+x^3+x^2+x$ on a set of `points`, the evaluation function's second argument.

Next, the evaluation function and the variation operators are registered similarly to Figure 1, while line 14 to the end remain exactly the same. Furthermore, using external libraries such as NetworkX [5] and PyGraphviz [6], the best primitive trees can be visualized[1] as presented in Figure 2.

The primitives are not limited to standard library operators, any function or instance method can be added to a primitive set. Terminals can be any type of objects and even functions without argument. The next example, presented in Figure 3, takes advantage of this flexibility and reduces the runtime of the previous example by vectorizing the evaluation using Numpy [7], a library of high-level mathematical functions operating on multidimensional arrays.

The idea is to evolve a program whose argument is a vector instead of a scalar. Most of the code remains identical, only minor modifications (highlighted in Figure 3) are required. First, we replace the operators in the primitive set by Numpy operators that work on vectors (lines 6 to 9). Then, we remove the loop from the evaluation function (line 12), since it is implicit in the operators. Finally, we replace the `sum` and `sqrt` functions by their faster Numpy equivalent (line 13) and our regression problem is now vectorized.

[1] See the notebook version of this article for the complete code to visualize the tree: http://github.com/DEAP/notebooks.

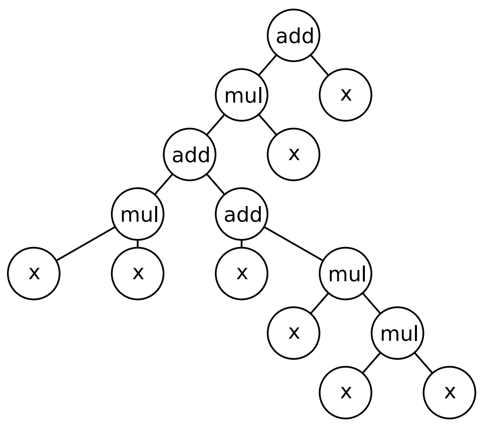


Fig. 2: Example of a GP individual generated with DEAP.

The execution is thereby significantly improved as the scalar example runs in around 3 seconds to optimize the regression on 20 points, while the vectorial runtime is identical but for a regression on 1000 points. By modifying only 6 lines of code, not only are we able to vectorize our problem, but the runtime is reduced by a **factor of 50**.

In addition to the wide support of function and object types, DEAP's *gp* module also supports automatically defined functions (ADF), strongly typed genetic programming (STGP), and object-oriented genetic programming (OOGP), for which examples are provided in the library documentation.

Distributed Island Model

The island model paradigm consists in multiple populations evolving separately and exchanging individuals on a regular basis. The final example illustrates how this scheme can be implemented with DEAP and SCOOP. The code presented in Figure 4 evolves 5 islands of 300 individuals. The algorithm runs for 40 generations, and every 10 generations, the 15 best individuals from one island are migrated to the next, following a ring topology.

```
import numpy
from deap import algorithms, base, creator, tools, gp

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Tree", gp.PrimitiveTree, fitness=creator.FitnessMin)

pset = gp.PrimitiveSet(name="MAIN", arity=1)
pset.addPrimitive(numpy.add, arity=2)
pset.addPrimitive(numpy.subtract, arity=2)
pset.addPrimitive(numpy.multiply, arity=2)
pset.addPrimitive(numpy.negative, arity=1)

def evaluateRegression(individual, points, pset):
    func = gp.compile(expr=individual, pset=pset)
    sqerrors = (func(points)-(points**4 + points**3 +
                              points**2 + points))**2
    return (numpy.sqrt(numpy.sum(sqerrors) / len(points)),)

toolbox = base.Toolbox()
toolbox.register("expr", gp.genFull, pset=pset, min_=1, max_=3)
toolbox.register("individual", tools.initIterate, creator.Tree,
                 toolbox.expr)
toolbox.register("population", tools.initRepeat, list,
                 toolbox.individual)
toolbox.register("evaluate", evaluateRegression,
                 points=numpy.linspace(-1, 1, 1000), pset=pset)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut,
                 pset=pset)
toolbox.register("select", tools.selTournament, tournsize=3)

if __name__ == "__main__":
    pop = toolbox.population(n=300)
    algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40)
    print(tools.selBest(pop, k=1)[0])
```

Fig. 3: Vectorized Genetic Programming example.

The for-loop starting at line 6 maps the algorithm to each island (line 7), making them evolve independently for FREQ=10 generations. Then, the resulting populations are recovered on line 8 and a ring topology migration is applied on line 9, using the built-in migRing operator to exchange individuals between islands. Since eaSimple uses the operator registered on line 1 to map the evaluation on the individuals, fitnesses are also computed in parallel. Therefore, the computations are distributed at 2 different levels.

The distribution scheme is presented in Figure 5. Each island evolution is executed by a distinct process and so is every evaluation. The listing in Figure 4 is generic. It could thus replace the main section of any of the previously presented examples: lines 16 and 17 in Figure 1(a) and lines 24 and 25 in Figure 3. To run this code on multiple processors, assuming the source code is in file island.py, one simply needs to enter the command line: `python -m scoop island.py`

Evolution Support

DEAP comes with several supporting tools that can be easily integrated into any algorithm. This section presents some of them in the context of the OneMax example (Figure 1).

The first tool, *Statistics*, computes statistics on arbitrary attributes of designated objects, usually the fitness of the individuals. The attribute is specified by a key function at the statistics object instantiation before starting the algorithm, between lines 16 and 17 of Figure 1(a).

```
stats = tools.Statistics(key=operator.attrgetter("fitness.values"))
```

This is followed by the registration of the statistical functions as for a toolbox.

```
stats.register("avg", numpy.mean)
stats.register("min", numpy.min)
stats.register("max", numpy.max)
```

Ultimately, at every generation, a statistical record of the population is compiled using the registered functions.

```
record = stats.compile(pop)
print(record)
```

```
1 toolbox.register("map", futures.map)
2 toolbox.register("migrate", tools.migRing, k=15,
↪                 selection=tools.selBest)
3 NGEN, FREQ = 40, 10
4 toolbox.register("algorithm", algorithms.eaSimple, toolbox=toolbox,
↪                 cxpb=0.5, mutpb=0.2, ngen=FREQ, verbose=False)
5 islands = [toolbox.population(n=300) for i in range(5)]
6 for i in range(0, NGEN, FREQ):
7     results = toolbox.map(toolbox.algorithm, islands)
8     islands = [island for island, logbook in results]
9     toolbox.migrate(islands)
```

Fig. 4: Distributed Island Model example.

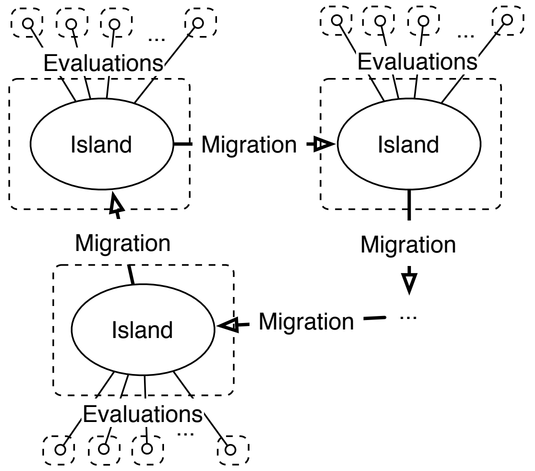


Fig. 5: Island model distribution scheme. Each island runs in a different process and evaluations are also done in parallel. SCOOP takes care of all necessary load balancing.

The statistics compilation produces a dictionary containing the statistical keywords and their respective value. These last lines, added after the evaluation part of Figure 1(b) (line 34), will produce a screen log of the evolution statistics.

For posterity and better readability, statistics can also be logged in a *Logbook*, which is simply a list of recorded dictionaries that can be printed with an elegant layout. For example, the following lines create a new logbook, then record the previously computed statistics and print them to the screen.

```
logbook = tools.Logbook()
logbook.record(gen=g, nevals=300, fitness=record)
print(logbook)
                fitness
            ---------------
gen nevals  avg     min max
0   300     49.9933 35  64
```

The next tool, named *Hall of Fame*, preserves the best individuals that appeared during an evolution. At every generation, it scans the population and saves the individuals in a separate archive that does not interact with the population. If the best solution disappears during the evolution, it will still be available in the hall of fame. The hall of fame can be provided as an argument to the algorithms (Figure 1(a) line 17) as follows:

```
halloffame = tools.HallOfFame(maxsize=10)
algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40,
                    halloffame=halloffame)
```

Moreover, the hall of fame can be updated manually right after the population is evaluated (Figure 1(b) line 34) with the following line of code.

```
halloffame.update(pop)
```

The hall of fame proposes a list interface where the individuals are sorted in descending order of fitness. Thus, the fittest solution can be retrieved by accessing the list's first element.

```
best = halloffame[0]
```

A Pareto dominance version of the hall of fame is also available. The *Pareto Front* maintains an archive of non-dominated individuals along the evolution. Its interface is the same as the standard hall of fame.

Another tool, called the *History*, tracks the genealogy of the individuals in a population. By wrapping the variation operators, the history saves the parents of each individual. This feature is added to the variation operators of the toolbox with the following lines.

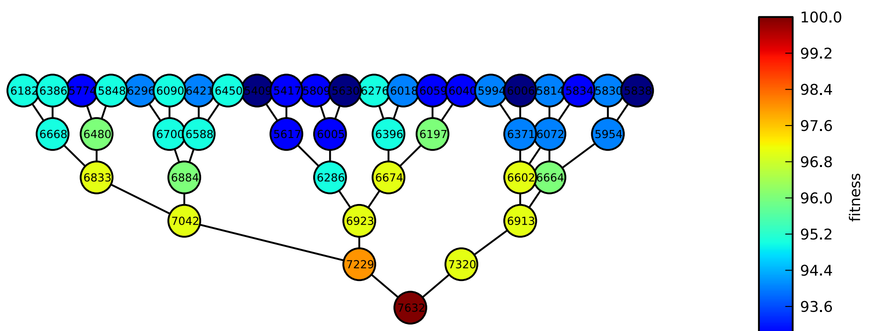


Fig. 6: Genealogy tree of the best individual found in the OneMax problem during the last 5 variation operations. The ancestors are at the top while the fittest offspring is at the bottom. The numbers represent the solution index. A node with two incoming links is the result of a crossover (see 7229), while a node with a single incoming link comes from a mutation (see 7320).

```
history = tools.History()
toolbox.decorate("mate", history.decorator)
toolbox.decorate("mutate", history.decorator)
```

It is therefore possible to determine the genesis of individuals. Figure 6 presents the genealogy of the best individual in the OneMax example for the last 5 variation operations. The graph is produced by the NetworkX library and the following listing.

```
h = history.getGenealogy(haloffame[0], max_depth=5)
graph = networkx.DiGraph(h)
networkx.draw(graph)
```

The last presented tool is a checkpointing facility. Rather than a DEAP object, checkpointing is ensured by the powerful *pickle* standard library module that can serialize almost any Python object. Checkpointing only requires selecting objects that shall be preserved and the write frequency. This is exactly what is done in the following lines that can be added at the end of the generational loop of Figure 1(b).

```
import pickle

if g % freq == 0:
    cp = dict(population=pop, generation=g, rndstate=random.getstate())
    pickle.dump(cp, open("checkpoint.pkl", "w"))
```

These last lines write into a file the population, the generation number, and the random number generator state so that this information can be used later to restart an evolution from this exact point in time. Reloading the data is as simple as reading the pickled dictionary and accessing its attributes.

```
cp = pickle.load(open("checkpoint.pkl", "r"))
pop = cp["population"]
g = cp["generation"]
random.setstate(cp["rndstate"])
```

This simple mechanism provides fault tolerance to any sort of evolutionary algorithms implemented with DEAP. This happens to be critical when exploiting large computational resources where chances of failure grow quickly with the number of computing nodes. Even in very stable execution environments, checkpoints can significantly reduce the amount of time spent experimenting by allowing evolutions to restart and continue beyond the original stopping criteria.

Conclusion

DEAP proposes an agile framework to easily prototype and execute explicit evolutionary algorithms. Its creator module is instrumental for building custom transparent data structures for the problem at hand. Its toolbox gathers all necessary operators and their arguments in a single handy structure. Its design provides straightforward distributed execution with multiple distribution libraries. The presented examples covered only a small part of DEAP's capabilities that include evolution strategies (including CMA-ES), multi-objective optimization (NSGA-II and SPEA-II), co-evolution, particle swarm optimization, as well as many benchmarks (continuous, binary, regression, and moving peaks), and examples (more than 40).

After more than 4 years of development, DEAP version 1.0 has been released in February 2014. DEAP is an open source software, licensed under LGPL, developed primarily at the Computer Vision and Systems Laboratory of Université Laval, Québec, Canada. DEAP is compatible with Python 2 and 3. It has a single dependency on Numpy for computing statistics and running CMA-ES. Try it out and become nimbler too: http://deap.gel.ulaval.ca.

References

[1] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[2] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné. DEAP: A Python Framework for Evolutionary Algorithms. *In Companion Proceedings of the Genetic and Evolutionary Computation Conference*, pages 85–92, 2012.

[3] Y. Hold-Geoffroy, O. Gagnon, and M. Parizeau. SCOOP: Scalable COncurrent Operations in Python. http://www.pyscoop.org/

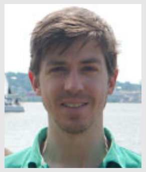
[4] J. R. Koza. Genetic Programming - On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.

[5] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. *In Proceedings of the Python in Science Conference*, pages 11-15, 2008. http://networkx.github.io

[6] A. A. Hagberg, D. A. Schult, and M. Renieris. PyGraphviz a Python interface to the Graphviz graph layout and visualization package. http://networkx.lanl.gov/pygraphviz

[7] E. Jones and T. Oliphant and P. Peterson and others. SciPy: Open source scientific tools for Python. http://www.scipy.org

About the authors

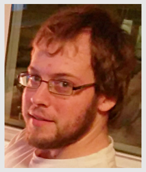


François-Michel De Rainville received his Master's degree from Université Laval, Québec, Canada, in 2010, for his work on interactive designs of experiments to enhance comprehension of complex systems. He is currently pursuing his PhD degree at Université Laval on using a swarm of robots to explore and analyze unknown environments. His major area of interest are robotics, pattern recognition, machine learning, evolutionary algorithms, and computer vision. He is one of the main DEAP developers.

Homepage: http://vision.gel.ulaval.ca/~fmdrainville
Email: francois-michel.de-rainville.1@ulaval.ca

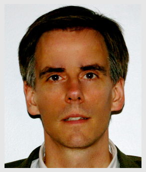
Félix-Antoine Fortin received a M.Sc. in Electrical Engineering from Université Laval in 2010 for his work on automatic surveillance camera placement with genetic algorithms. He is currently completing a Ph.D at Université Laval on multimodal optimization while working as an high performance computing analyst at Calcul Québec, a research consortium for High Performance Computing (HPC). His main research interests are optimization, pattern recognition, evolutionary algorithms, and distributed computing. He is also one of the main developer of DEAP.

Homepage: http://vision.gel.ulaval.ca/~fafortin/
Email: felix-antoine.fortin.1@ulaval.ca



Marc-André Gardner received a B.Sc. in Computer Engineering from Université Laval in 2012. He has worked on bloat control in genetic programming, and is currently completing a M.Sc. in Electrical Engineering on stochastic grammar optimization applied to genetic programming. He has also worked on a task distribution framework in Python, and is a major contributor to DEAP, in addition to being one of its power users.

Email: marc-andre.gardner.1@ulaval.ca



Marc Parizeau is a professor of Computer Engineering at Université Laval, Québec City. He obtained his Ph.D. in 1992 from École Polytechnique de Montréal. His research interests are mainly in the field of intelligent systems, in machine learning for pattern recognition in particular, as well as in parallel and distributed systems. In 2008, he created a High Performance Computing (HPC) center at Université Laval, and is the current scientific Director of Calcul Québec, an HPC consortium for the province of Québec, also one of the four regional divisions of Compute Canada, the national HPC platform.

Homepage: http://vision.gel.ulaval.ca/~parizeau
Email: parizeau@gel.ulaval.ca

Christian Gagné received a B.Ing. in Computer Engineering and a PhD in Electrical Engineering from Université Laval in 2000 and 2005, respectively. He is professor of Computer Engineering at Université Laval since 2008. His research interests are on the engineering of intelligent systems, in particular systems involving machine learning and evolutionary computation. He is member of editorial board of the Genetic Programming and Evolvable Machines journal, and participated to the organization of several conferences. He is also the main developer of Open BEAGLE, a generic C++ framework for evolutionary computation, from which lessons learnt served as inspiration to the design of DEAP.

Homepage: http://vision.gel.ulaval.ca/~cgagne
Email: christian.gagne@gel.ulaval.ca