DEAP: A Python Framework for Evolutionary Algorithms

François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner
Marc Parizeau, and Christian Gagné
Laboratoire de vision et systèmes numériques
Département de génie électrique et de génie informatique
Université Laval, Québec (Québec), Canada G1V 0A6
{francois-michel.de-rainville.1, felix-antoine.fortin.1, marc-andre.gardner.1}@ulaval.ca,
{marc.parizeau, christian.gagne}@gel.ulaval.ca

ABSTRACT

DEAP (Distributed Evolutionary Algorithms in Python) is a novel evolutionary computation framework for rapid prototyping and testing of ideas. Its design departs from most other existing frameworks in that it seeks to make algorithms explicit and data structures transparent, as opposed to the more common black box type of frameworks. It also incorporates easy parallelism where users need not concern themselves with gory implementation details like synchronization and load balancing, only functional decomposition. Several examples illustrate the multiple properties of DEAP.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*heuristic methods*; D.2.11 [**Software**]: Software Architectures—*domain-specific architectures*

Keywords

Parallel Evolutionary Algorithms, Software Tools

1. INTRODUCTION

Over the years, several object-oriented programming tools have been developed for Evolutionary Computation (EC), for example EO [8], ECJ [9], and Open BEAGLE [6], among many others. Albeit in different ways, they all implement somewhat complex low level mechanisms that allow development of higher level Evolutionary Algorithm (EA). They usually provide a fairly large library of common EA components, most often in different flavours, eventually evolving into what is commonly called a black box framework [10], that provides high level functionality while trying to hide implementation detail as much as possible. They may be very generic in nature, but a user who wishes to implement a new type of EA, or even a new variation on a common type, is most often faced with the daunting task of understanding intricate details that in effect hinders his creativeness and productivity. Of course, framework authors will argue that "their" framework is better, friendlier, more generic, or perhaps more powerful than others, but we argue that they are all way too complex and bloated, and not well enough documented, including our own Open BEAGLE. Code bloat and complexity makes documentation all the more painful; poor documentation emphasizes bloat and complexity.

Developing reusable frameworks usually starts by analyzing the problem domain and solving a few applications within this domain [10]. The process then evolves into a white box framework where common functions are encapsulated into classes and applications are built using inheritance. With time, higher level functions are added and internal details are gradually hidden, eventually morphing the white box into a black box. This is all fine in many cases: with added functionality, users no longer need to worry about internal structures and life is made much easier, that is as long as the needed features are present in the framework. However, in the context of EC, the opaqueness of black boxes can become a major hurdle especially when researching for new algorithms or tackling special problems that require custom algorithms. Lacking adequate documentation, the programmer needs to look into the gory details of lower level mechanisms and, depending on his prior experience, may or may not achieve his goals in a reasonable amount of time. At this point, it is important to remember that EAs form quite a diverse field. They are all based on common principles, but they vary in so many ways that no black box framework will ever offer all of the functions required by its potential users.

With DEAP (Distributed Evolutionary Algorithms in Python), our aim is to provide a toolbox that encourages users to write their own evolutionary algorithms, explicitly controlling every aspect of the evolutionary process: data types, fitness measures, population initialization, operators, evolutionary loop, etc. In particular, we want the users to write their own evolutionary loops where every step is explicit, and we want these loops to be short, easy to read/understand, and easy to document.

Another fundamental objective behind DEAP is to provide transparent parallelism, as much as possible. Researchers want to experiment with new EC ideas to solve complex real-world problems. Their foremost worry is to validate in a reasonable amount of time whether or not these ideas are any good. With the stagnation of processor clock frequencies and the usual high complexity of real-world problems, parallelism is thus a fundamental requirement. But parallel programming is hard! You need to identify oppor-

GECCO'12, July 7-11, 2012, Philadelphia, Pennsylvania, USA.

tunities for parallelism, which is usually fairly easy for EAs, but you also have to manage communication and synchronization between processes, to balance loads between processors, and to squash ugly bugs that inevitably tend to creep up from every corner of a parallel application. In this context, DEAP's strategy is to leave the easier task of specifying what should be run in parallel to the user, but essentially takes care of all of the rest.

To achieve its goals, DEAP builds on the Python programming language for its coherent syntax and its many powerful features. Namely, its full support of object-oriented programming, its pure dynamical runtime environment, as well as its good support of the functional programming paradigm, which makes it close to ideal for EAs that are intrinsically functional in nature. Of course, there is also a down side: being interpreted, Python can lead to slow execution compared to other compiled languages. But this limitation can be mitigated by its rapid prototyping capabilities, and the fact that Python is easily interfaced to other languages like C or C++. The strategy is thus to first develop prototypes in pure Python to test ideas, and later, if these ideas are shown worthwhile, to recode in C (or any other similar language) only those parts of the EA that are particularly compute intensive like, for instance, fitness evaluation or any sophisticated operators like some multi-objective selection (e.g., domination-based sorting of NSGA-II). It should be noted that basic operations like sorting elements of containers are already implemented in Python through tight integration with C. Also, Python is well known for its large library of extension modules that link to efficient C or FORTRAN libraries. In particular the combination of the NumPy[1], SciPy[1], and matplotlib[2] Python modules can advantageously replace commercial rapid prototyping tools like Matlab for scientific computations.

The rest of this paper is organized as follows. Sec. 2 starts with an overview of current existing frameworks, their black box implementation and some of their limitations according to what we propose with our model. Next, Sec. 3 summarizes the basic concepts underlying the development of DEAP as a white box optimizer, and Sec. 4 presents the framework in details with three different code examples. Finally, Sec. 5 presents different tools provided with DEAP to ease analysis of algorithms and allow a very easy parallelization of the computations.

2. RELATED WORK

Robert and Johnson [10] define a good framework as: "[...] simple enough to be learned, yet must provide enough features that it can be used quickly and hooks for the features that are likely to change." This definition and our experience developing the Open BEAGLE framework bring us to the conclusion that the usual EC framework development model has some limiting flaws. In the present section, we identify those limitations in order to demonstrate the validity of DEAP's design choices.

Most popular EC frameworks propose a large set of ready made algorithms. The usefulness of having many implemented algorithms is undeniable. However, the complexity of the mechanisms that keep genericity in those algorithms is often overwhelming. Adding or modifying an algorithm of the framework is therefore not an easy task for a casual user. The problem is that, nowadays, most real world problems require some modifications to the original algorithms in order to achieve the next level of performance. Most users do not have the time or resources to understand all the mechanisms included in the algorithms in order to extend them.

To compensate for the opacity of the algorithm implementations, some frameworks propose alternative higher level programming language. ECJ proposes to configure the algorithms via a pipeline principle submitted to the framework through a configuration file coded in the INI format. Open BEAGLE offers the possibility to customize the algorithm workflow and parameters via an XML configuration file. EASEA and Guide [3] implement a high level programming language for EO and a graphical interface that produces C++ code compatible with EO. From a software engineering point of view, those solutions are quite interesting and their design are hard to argue with. They can help users to adapt the algorithms to better fit their problem's requirements, as long as the needed functions are either present in the framework or allowed by the abstraction language. However, from this point of view, developers of black box frameworks are left with the impossible task of anticipating every possible extension of the original algorithms.

Parallel evolutionary algorithms (PEAs) is also a blooming field in itself. Alba and Tomassini [1] divided PEAs in two categories: structured EAs and global parallelism (also called master-slave). The former implies the implementation of specific algorithms that modify the typical canonical algorithms by either distributing the populations or the individuals according to the underlying hardware architecture to achieve better performance. The latter offers to keep the logic of the algorithm untouched and to distribute the operations across all computing units. Each method suffers from drawbacks. Structured EAs are not well adapted to massive parallel architectures, while global parallelism generally involves more communications that induce bottlenecks and limit performance [2, 5].

3. DESIGN PRINCIPLES

Because EC is a sophisticated field with very diverse techniques and mechanisms, even well designed evolutionary frameworks can become very complicated under the hood. And the more complicated they become, the less likely the commoner will ever take a peek under the hood to consider making modifications. But research in EC, or simply using EC for solving real-world problems, requires making changes most of the time. Thus, DEAP's approach is like removing the hood altogether, effectively forcing users to look constantly at most of the internal components of the car, helping them become good mechanics, capable of making repairs themselves, eventually leading them to start thinking about building custom roadsters!

DEAP is a lightweight framework that focuses on providing both basic EC operators (parts) and general mechanisms to easily build custom parts to implement sophisticated EAs. To achieve this, it uses the powerful Python scripting language to provide the essential glue for assembling EA parts into coherent EC systems. Below are the five founding hypotheses of DEAP:

HYPOTHESIS 1. *The user knows best. Users should be able to understand the internal mechanisms of the frame-*

[1] `http://numpy.scipy.org`
[2] `http://matplotlib.sourceforge.net`

work so that they can extend them easily to better suit their specific needs.

HYPOTHESIS 2. *User needs in terms of algorithms and operators are so vast that it would be unrealistic to think of implementing them all in a single framework. However, it should be possible to build basic tools and generic mechanisms that enable easy user implementation of most EA variant.*

HYPOTHESIS 3. *Speedy prototyping of ideas is often more precious than speedy execution of programs. Moreover, code compactness and clarity are also very precious.*

HYPOTHESIS 4. *Even though interpreted, Python is fast enough to execute EAs. Whenever execution time becomes critical, compute-intensive components can always be recoded in C. Many efficient numerical libraries are already available through Python APIs.*

HYPOTHESIS 5. *Easy parallelism can alleviate slow execution.*

And these hypotheses lead to the following objectives:

1. **Rapid prototyping.** Provide an environment allowing users to quickly implement their own algorithms without compromise.
2. **Parallelization made easy.** Allow straightforward parallelization; users should not be forced to specify more than the granularity level of their functional decomposition.
3. **Preach by examples.** Although the aim of DEAP itself is not to provide ready made solutions, it should nevertheless come with a substantial set of real-world examples to guide the users apprenticeship.

4. CORE ARCHITECTURE

The core architecture of DEAP is built around different components that define the specific parts of what is an evolutionary algorithm. Fig. 1 shows the principal modules forming the framework. DEAP's core is composed of three modules: *base*, *creator*, and *tools*.

The *base* module contains objects and data structures frequently used in EC that are not already implemented in the Python standard library. Python providing most of the data structures required, this module actually implements only three classes: a generic fitness, a prefix coded tree and a toolbox. The toolbox is a container for the tools (operators) that the user wants to use in his EA. For instance, if the user needs a mutation in his algorithm, but has access to several mutation designs, he will choose the one model best suited for his current problem, say "MutationXYZ", and register it into the toolbox with a generic "mutation" alias. In this way, he will be able to build algorithms that are decoupled from operator sets. If he later decides that some other mutation is better suited, his algorithm will remain unchanged, he will only need to update the toolbox used by the algorithm. The concept underlying the toolbox can be found in most blackbox frameworks. However, DEAP's toolbox distinguishes itself on two aspects. First, other frameworks usually force a specific signature for every type of operation to allow operator swapability. In DEAP, the signature

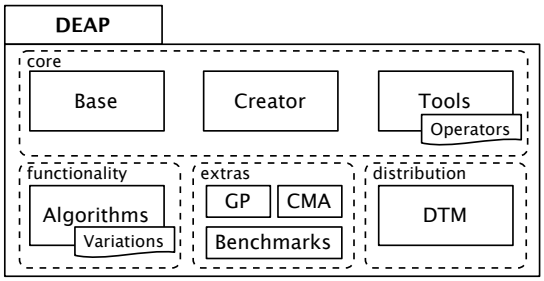

Figure 1: Modules of the core architecture.

is enforced by no means, hence allowing the user to implement some special operator as he wants without having to circumvent implementation intricacies. Second, blackboxes usually provide default values for most of the parameters to simplify operator calling. This practice contradicts our first hypothesis, and thus we do not define default value for any operator parameter. The toolbox performs this simplification task by allowing the user to register parameter values with the operator. This way, the user has to understand every operator he uses, and the parameters values are always explicitly stated before the algorithm definition, therefore avoiding any possible ambiguity.

The *creator* module is a meta-factory that allows creation of classes via both inheritance and composition using a functional programming paradigm, therefore liberating the user from the burden of class definition. Attributes, both data and functions, can be dynamically added to create new object classes empowered by the user to provide user specific EA functionalities.

The *tools* module contains frequently used EA operators. It also provides objects that ease the different analysis tasks of EAs such as checkpointing, statistics computation, and genealogy. These will be described in detail in Sec. 5.

The core functionalities of DEAP are levered by the *algorithms* module that contains four commonly used algorithms in EC: generational, (μ, λ), $(\mu + \lambda)$, and *ask-and-tell* [4]. However, DEAP is not limited in any way to these four. They are only a starting point for users to develop their own customized algorithms.

Operators and tools not provided by the core modules generally have their own module, like Genetic Programming (GP) operators and data structures can be found in the *gp* module, and CMA-ES in the *cma* module. The *benchmarks* module includes different state-of-the-art benchmark functions that can be used to assess algorithm performances.

The last module of the framework, named *dtm* for Distributed Task Manager, handles parallelism. It will be described in detail in Sec. 5.3.

To explain how these modules interact, we now present three examples:

Example 1. The first example illustrates how the one-max problem can be solved with DEAP. All code lines are in real order of appearance in the original script and no line has been omitted.

First, we start by importing the necessary modules.

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

Next, the fitness evaluation function is defined as the sum of ones within an individual using the Python standard `sum` function that sums the elements of any iterable object. In DEAP, evaluation functions always return a tuple of values even for single-objective problems, because they are treated as a special case of multi-objective problems.

```
3 def evalOneMax(individual):
4     return (sum(individual),)
```

Thereafter, we use the creator to create a `FitnessMax` class encapsulating our fitness values and producing fitness maximization. To do this, we make this new class inherit from `base.Fitness` and we set its `weights` attribute accordingly.

```
5 creator.create("FitnessMax", base.Fitness,
  ↪ weights=(1.0,))
```

The `create` function of the `creator` module expects at least two arguments. The first (`FitnessMax`) is the name of the new class, while the second (`base.Fitness`) is the parent class. Any additional argument given to this function is added to the new class as an attribute. In this case, the `weights` attribute is a tuple containing a single positive unit weight achieving fitness maximization. To minimize the fitness, we would use -1 instead of 1. For multi-objective optimization, we would assign a tuple with as many weights as there are objectives.

The second built class encapsulates our individuals. Here, for the one-max problem, it inherits from the Python standard list, and it has a `fitness` attribute of the just created `FitnessMax` type.

```
6 creator.create("Individual", list, fitness=
  ↪ creator.FitnessMax)
```

Because it is so simple to create a class using the creator, DEAP does not propose any basic individual type. Users are asked to design their own types based on their specific needs.

Once types are created, we instantiate a new toolbox.

```
7 toolbox = base.Toolbox()
```

The registration of functions (tools) in the toolbox works as follows: the first argument corresponds to the alias of the registered function, the second argument is the function itself, and subsequent arguments are automatically passed to the function when the alias is called. The `Individual` class has no initialization method, meaning that any new instance of this class is an empty list with a `FitnessMax` attribute. In order to ease individuals initialization, we register a `bit` function as an alias for the standard Python random integer generator.

```
8 toolbox.register("bit", random.randint, 0, 1)
```

When called, this `bit` function returns a random integer, either 0 or 1. Next, we register an `individual` alias that acts as an initialization operator for the individuals.

```
9 toolbox.register("individual", tools.initRepeat,
  ↪ creator.Individual, toolbox.bit, n=100)
```

The function assigned to this alias is `tools.initRepeat` which takes three arguments: the first one is a data structure constructor, in this case it is our individual constructor `creator.Individual`, the second one is the function used to generate the content filling for that data structure, and the last one is the number of elements to generate. The registered `individual` function is then able to generate individuals composed of 100 random bits. In a similar fashion, we register an operator named `population` capable of generating a list of 300 individuals.

```
10 toolbox.register("population", tools.initRepeat,
   ↪ list, toolbox.individual, n=300)
```

Finally, additional operators are registered under meaningful aliases recognized by the algorithms provided by the `algorithms` module.

```
11 toolbox.register("evaluate", evalOneMax)
12 toolbox.register("mate", tools.cxTwoPoints)
13 toolbox.register("mutate", tools.mutFlipBit,
   ↪ indpb=0.05)
14 toolbox.register("select", tools.selTournament,
   ↪ tournsize=3)
15 toolbox.register("map", map)
```

The one-max evaluation function becomes `evaluate`, the two points crossover (`cxTwoPoints`) aliases to `mate`, the flip bit mutation (`mutFlipBit`) with an independent probability (`indpb`) of application on each element of 5% becomes `mutate`, and the tournament selection (`selTournament`) between 3 participants (`tournsize`) turns into `select` The final registered operator `map` is a place holder. It defaults to the standard `map` function of Python which applies a function to every element of an iterable object and returns a list of the results. As shown in Sec. 5.3, by replacing this mapping tool with `dtm.map`, the user automatically benefits from the parallelization and load balancing features of DEAP.

The simple generational algorithm (`eaSimple`) defined in DEAP requires a population, a toolbox, and three parameters to run. These parameters are the crossover and mutation probabilities, and the number of generations. The population allocation is done by calling the appropriate toolbox function. Finally, the algorithm is launched and the final population is returned after 40 generations.

```
16 CXPB, MUTPB, NGEN = 0.5, 0.2, 40
17 pop = toolbox.population()
18 pop = algorithms.eaSimple(pop, toolbox, CXPB,
   ↪ MUTPB, NGEN)
```

Example 2. The first example showed typical initialization for a DEAP program, but using a canned EA. The second example keeps the same initialization but shows more details of the evolutionary loop. The algorithm invocation of Ex. 1 (line 18) is replaced below by explicit calls to operators.

First, the toolbox evaluation function is mapped to every individual of the population in order to compute their fitness (new line 18). Then, a loop assigns the returned fitness values to each individual (lines 19 & 20).

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

The evolutionary algorithm itself is a simple for loop with a termination criterion based on the user specified maximum number of generations (line 21).

```
21 for g in range(NGEN):
22     pop = toolbox.select(pop, k=len(pop))
23     pop = algorithms.varAnd(pop, toolbox, CXPB,
       ↪ MUTPB)
24     fits = toolbox.map(toolbox.evaluate, pop)
25     for ind, fit in zip(pop, fits):
26         ind.fitness.values = fit
```

On each cycle of the evolutionary process, the defined selection operator is applied on the population to select a new one (line 22). The number of selected individuals (`k`) corresponds to the size of that initial population, as returned by the standard `len` Python function. The `varAnd` function applies variation operators (those defined in the toolbox), namely crossover and mutation, and returns an updated population (line 23). The function signature is similar to the generational algorithm of Ex. 1, except for the number of generations which is now explicitly specified in the for loop. Finally, the resulting individuals are evaluated for the next iteration (lines 24 to 26).

The variation function (line 23) can also be exploded in a more explicit form. First, we proceed with the duplication of the population using the `clone` function of the toolbox.

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

Then, the crossover exploits the powerful slicing operator of Python to build consecutive pairs of potential children. A random number is drawn and if this number is less than the user specified crossover probability, the children are mated and their fitness is invalidated by deleting the associated values.

```
for child1, child2 in zip(pop[::2], pop[1::2]):
    if random.random() < CXPB:
        toolbox.mate(child1, child2)
        del child1.fitness.values
        del child2.fitness.values
```

Finally, a similar logic is applied to mutation. For every individual a random number is drawn and compared to the mutation probability, if the individual is mutated, its fitness is invalidated.

```
for mutant in pop:
    if random.random() < MUTPB:
        toolbox.mutate(mutant)
        del mutant.fitness.values
```

Example 3. Now that the foundations of the framework have been explained and understood, we can move on to a more ambitious problem, for instance, a modified version of the co-evolution of sorting networks as described by Hillis [7]. For this implementation, the hosts are represented by a list of pairs of wire index (comparators), while the parasites are lists of integers (later translated to binary strings). For example, $[(1, 3), (2, 4)]$ represents a network with two comparators, the first connecting wires 1 and 3, and the second connecting wires 2 and 4. For 4 inputs sorting networks, a parasite $[3, 8]$ represents two sequences to be sorted: $[0, 0, 1, 1]$ and $[1, 0, 0, 0]$. The two-objectives host optimization are to minimize the number of errors made by a network and to minimize its number of connectors. The single-objective parasite optimization is to maximize the number of errors made by the network it is evaluated against.

After module importation and constant definition, the evaluation function is defined to assess the fitness of a host against a parasite.

```
1 import random
2 from deap import algorithms, base, creator, tools
3 import sortingnetwork as sn
4
5 INIT_SIZE, INPUTS, MAXGEN = 40, 12, 250
6 H_CXPB, H_MUTPB = 0.5, 0.3
7 P_CXPB, P_MUTPB = 0.5, 0.3
8 def evalNetwork(host, parasite, dimension):
9     network = sn.SortingNetwork(dimension, host)
10    cases = [int2bin(case) for case in parasite]
11    return (network.assess(cases), len(host))
```

The network is created from the individual (line 9), and the parasite's test cases are translated from the list of integers to a sequence of binary strings (line 10) before the network performance is evaluated (line 11). This evaluation function has two objectives, the number of errors made by the network when sorting the test cases and the number of connectors used. Then, a non standard mutation operator is defined to work directly on the described host type. It replaces connectors by random ones generated by the function `comparator` given as argument.

```
12 def mutComparator(individual, comparator, indpb):
13     for indx in range(len(individual)):
14         if random.random() < indpb:
15             individual[indx] = comparator()
16     return (individual,)
```

As in Ex. 1, the creator is used to create the types for the problem we described.

```
17 creator.create("FitnessHost", base.Fitness,
   ↪ weights=(-1.0, -1.0))
18 creator.create("FitnessParasite", base.Fitness,
   ↪ weights=(1.0, 0.0))
19 creator.create("Host", list, fitness=creator.
   ↪ FitnessHost)
20 creator.create("Parasite", list, fitness=creator.
   ↪ FitnessParasite)
```

The host fitness is created in order to minimize both objectives, while the parasite fitness maximizes the first objective and ignores the second one (because of its null weight). Then, both the `Host` and the `Parasite` classes are created and associated with their respective fitness attributes.

After types creation, we instantiate two independent toolboxes to contain the operator set for each species (lines 21 & 27). The host toolbox (`htbx`) is set to create a population that is a list of individuals. The individuals are hosts initialized with `INIT_SIZE` comparators and the comparators are connected to 2 random wires. A population in the parasite toolbox (`ptbx`) is a list of individuals, which are parasites made of 200 integers drawn between 0 and $2^l - 1$, with l the number of inputs.

```
21 htbx = base.Toolbox()
22 htbx.register("wire", random.randint, 0,
   ↪ INPUTS - 1)
23 htbx.register("comparator", tools.initRepeat,
   ↪ tuple, htbx.wire, n=2)
24 htbx.register("individual", tools.initRepeat,
   ↪ creator.Host, htbx.comparator, n=INIT_SIZE)
25 htbx.register("population", tools.initRepeat,
   ↪ list, htbx.individual)
26
27 ptbx = base.Toolbox()
28 ptbx.register("integer", random.randint, 0,
   ↪ 2**INPUTS - 1)
29 ptbx.register("individual", tools.initRepeat,
   ↪ creator.Parasite, ptbx.integer, n=200)
30 ptbx.register("population", tools.initRepeat,
   ↪ list, ptbx.individual)
```

The evaluation function is only registered in the host toolbox since both species are evaluated at the same time. The crossover for hosts is an unaligned one point crossover that can change individuals length, while the mutation is the

mutComparator defined on line 12. Selection among hosts is made by NSGA-II. The parasite toolbox mates individuals using a standard one point crossover, it mutates them by changing some integers by other integers drawn uniformly between low and up, and the selection is made using a 3 participants tournament on the first objective.

```
31 htbx.register("evaluate", evalNetwork,
   ↪ dimension=INPUTS)
32 htbx.register("mate", tools.cxMessyOnePoint)
33 htbx.register("mutate", mutComparator,
   ↪ comparator=htbx.comparator, indpb=0.05)
34 htbx.register("select", tools.selNSGA2)
35
36 ptbx.register("mate", tools.cxOnePoint)
37 ptbx.register("mutate", tools.mutUniformInt,
   ↪ low=0, up=2**INPUTS - 1, indpb=0.05)
38 ptbx.register("select", tools.selTournament,
   ↪ tournsize=3)
```

The complete algorithm is almost as simple as the one of Ex. 2 even though we are manipulating two populations. First we create populations of 300 hosts and 300 parasites, using their respective initialization method registered in each toolbox.

```
39 hosts = htbx.population(n=300)
40 parasites = ptbx.population(n=300)
```

A hall-of-fame (Pareto front type) is also instantiated to keep track of the best nondominated networks to appear during the evolution.

```
41 pareto = tools.ParetoFront()
```

Before starting the evolution, the populations are evaluated. Again, the toolbox map function is used to apply the evaluation function on host-parasite pairs. In this case, we repeatedly match the host population first individual with the parasite population first individual, then the second with the second, etc., and use the corresponding tuples as arguments for the toolbox evaluate function calls. The fitness are then assigned to each host and parasite in the for loop and the pareto front is updated with the evaluated hosts.

```
42 fits = htbx.map(htbx.evaluate, hosts, parasites)
43 for h, p, fit in zip(hosts, parasites, fits):
44     h.fitness.values = p.fitness.values = fit
45
46 pareto.update(hosts)
```

The evolution loop consists of variating individuals in each population, evaluating the populations one against the other, and selecting the next generation individuals until a predefined number of generations is reached.

```
47 for g in range(1, MAXGEN):
48     hoff = algorithms.varOr(hosts, htbx,
       ↪ len(hosts), H_CXPB, H_MUTPB)
49     parasites = algorithms.varAnd(parasites,
       ↪ ptbx, P_CXPB, P_MUTPB)
50
51     fits = htbx.map(htbx.evaluate, hoff,
       ↪ parasites)
52     for h, p, fit in zip(hoff, parasites, fits):
53         h.fitness.values = p.fitness.values = fit
54
55     pareto.update(hoff)
56
57     hosts = htbx.select(hosts+hoff, k=len(hosts))
58     parasites = ptbx.select(parasites,
       ↪ k=len(parasites))
```

This complete example illustrates the expressive power of DEAP, and its coding elegance and simplicity, even for relatively complicated algorithms. With DEAP, it is possible to remove the hood covering even the most sophisticated EAs, admire the engine clarity, and customize its parts.

Currently, DEAP bundles more than 30 complete working examples of ECs ranging from the simple one-max problem with bit-string GA, to the Artificial Ant with GP, including real function parameter optimization with CMA-ES, PSO, and EDA, and multi-objective optimization with NSGA-II and SPEA-II. We believe that the best way to document our framework is by showing how to design transparent implementations of classical problems using well known EAs. An ever increasing number of public examples also helps spreading variations of EC optimizers in the community, making them available to practitioners of other domains that would wish to apply those techniques to their own problems.

5. COMPLEMENTARY TOOLS

The proposed framework also offers tools that ease the interpretation of the results produced by the evolutionary algorithms and reduce the execution time by allowing easy distribution of tasks.

5.1 Best-of-Run

A typical need for the evolutionary practitioner is to keep the best individuals found along the evolution. DEAP proposes two object classes: the HallOfFame keeps the N best individuals found for a single-objective problem and the ParetoFront keeps the individuals along the first Pareto front of the entire evolution. Both objects have the same interface that only requires the user to provide a single list of individuals. Here is the creation and the usage of an hall-of-fame object in the context of Ex. 1, where we only want to keep the best individual of the run:

```
hof = tools.HallOfFame(maxsize=1)
[...]
hof.update(pop)
```

These objects provide a list interface where every element is always sorted in descending order, therefore the best individual of the previous example can be retrieved with hof[0].

5.2 Checkpointing

Checkpointing is made available in DEAP via a Checkpoint class. A checkpoint is a snapshot of the evolution environment that allows to restart the evolution from a particular state. The user registers the objects that he wants to checkpoint and dumps their state when needed. In the following we add the population of Ex. 1 to a checkpoint object under the name "population" and dump the checkpoint to a file. The checkpoint is then reloaded and the stored population is accessed by its registered name.

```
cp = tools.Checkpoint()
cp.add("population", pop)
cp.dump(open("checkpoint", "w"))
[...]
cp.load(open("checkpoint", "r"))
pop = cp["population"]
```

5.3 Distribution

In order to allow easy parallelization of specific parts of the user's algorithm, DEAP provides a Distributed Task

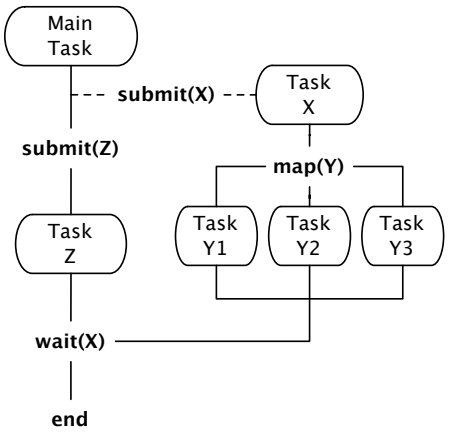

Figure 2: Example of task spawning within DEAP.

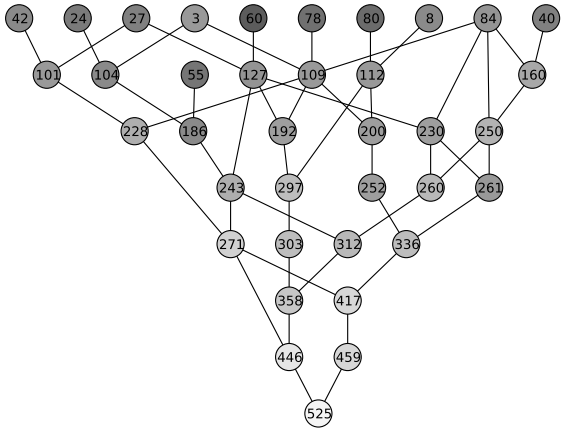

Figure 3: Genealogy tree of the best individual, lighter node colour means better individual.

Manager (DTM) module that can handle parallel sub-task creation and execution on both multi-core computers and clusters of networked nodes by relying on MPI. The DTM interface is composed of two main functions: `submit` which is used to execute another function as a single parallel sub-task; and `map` which is used to apply in parallel a given function to every element of an iterable object (e.g. a list). Its usage is almost completely transparent. For instance, in the previous examples, the only required changes in order to distribute the fitness evaluation tasks are to, somewhere during the initialization steps, import the *dtm* module and replace the map function in the toolbox with the `dtm.map` function.

```
from deap import dtm
toolbox.register("map", dtm.map)
```

No other change is necessary, DTM takes care of the rest. The map function call for the fitness evaluation of individuals (as seen in Ex. 2 and 3) will spawn parallel sub-tasks distributed by DTM across the worker nodes. DTM even load-balances them if they are not of equal duration. Any sub-task topology is possible, e.g. sub-tasks running on given worker nodes can spawn new sub-sub-tasks that will also get distributed and load balanced among all worker nodes, without any centralized intervention. Fig. 2 illustrates how a task can spawn sub-tasks which may themselves spawn sub-sub-tasks. Sub-task spawning can be blocking or non-blocking. A blocking spawn (continuous line) will halt the parent task until the child task returns with an answer, while a non-blocking spawn (dashed line) will return immediately to allow the parent to continue its execution until it is ready to process the result of its children.

In DTM, there is no hierarchy between worker nodes. Each worker manages two queues of tasks: those that are awaiting execution, and those that are halted, awaiting results from child tasks. When spawned, a task awaits execution in the queue of the node where it was created, but it can also be dynamically pushed to another node by the local load balancer. No matter where it is executed, the result of a task always returns to the node where it was originally spawned. Indeed, this is where the parent task is either running or halted, awaiting for its children's results (once started, a task may no longer migrate).

The load balancing mechanisms in DTM are fully decentralized. Every worker caches the load of others through epidemiological propagation; each time a worker communicates with a co-worker, for example to push a task or to return a task result, it also transmits its current load estimate. The load of a worker is assumed proportional to the total running time of all tasks that are currently assigned to this worker (waiting for execution, executing, or waiting for child results). The objective of the load balancer is to evenly distribute the load between worker nodes. The most underloaded workers transmit more often their load estimates to the overloaded ones, so that they inform the later of the relative under utilization of the former. For their part, the most overloaded workers transfer some of their tasks to the underloaded ones in order to reduce their own load.

5.4 History

The *History* tool helps keeping track of the genealogy of an evolution. In fact, it gathers the information produced by variations and compiles the individuals genealogy. Simply by decorating the registered variation operators, it is possible to memorize the parents and offspring of every solution produced during an evolution.

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

Then one can trace the evolution of the best individual of the population as in Figure 3, where individual 525 is the result of many combinations and mutations of the other individuals since the begin of the evolution, by simply retrieving the genealogy associated to the best individual :

```
import networkx
from matplotlib.pyplot import show

gen_best = history.getGenealogy(hof[0])
graph = networkx.DiGraph(gen_best).reverse()
networkx.draw(graph)
show()
```

5.5 Statistics

The *Statistics* object allows the user to register, as in the toolbox, statistic operations that should be performed at

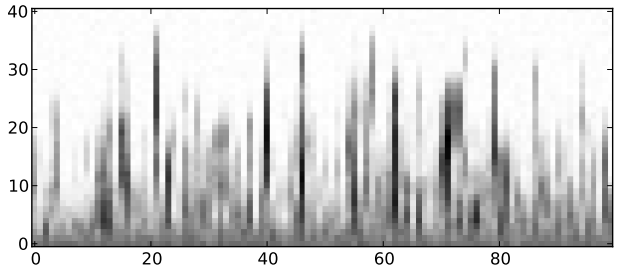

Figure 4: One-max gene average values heatmap.

every generation. The statistics are computed on an arbitrary attribute of a user designated object, most generally the individual fitness. A statistics object is associated to an attribute using a key in the following fashion:

```
stats = tools.Statistics(key=lambda ind: ind.
↪ fitness.values)
```

Then, statistical operations are registered under a specific alias. Some basic functions, like mean or standard deviation, are provided by the tools module.

```
stats.register("max", max)
stats.register("mean", tools.mean)
```

Once the operations registered, the statistics can be computed on a list of elements that possess the keyed attribute; in this case the population of individuals.

```
stats.update(pop)
```

Finally, the statistics object presents a two dimensional matrix interface to retrieve the computed statistics. The first dimension is the index of the list (mostly used when computing statistics of multiple demes with one statistics object), and the second dimension is the generation.

To demonstrate that *Statistics* can serve different purposes, here is the example of a heatmap for the average gene value of individuals in Ex. 1:

```
stats = tools.Statistics()
stats.register("avg", tools.mean)
```

If no key is provided, the statistics are computed directly on the elements of the list, therefore in this case on the individual genes. We use matplotlib to produce a simple heatmap:

```
from matplotlib.pyplot import imshow, show
imshow(stats.avg[0], origin="lower")
show()
```

The result is presented in Fig. 4. The x-axis represents the index of the gene and the y-axis the generation. White genes have an average value of 1 and black genes have an average value of 0. We observe that the evolution makes the average of all genes converge from 0.5 to 1.

6. CONCLUSION

Current major EC frameworks all do a good job of offering generic tools to solve hard problems using EAs. However, their implementation intricacies make them hard to extend for the commoner. Even experts can become overwhelmed when trying to implement special features. This paper has described a novel framework named DEAP, that combines the flexibility and power of the Python programming language with a clean and lean core of transparent EC components that facilitate rapid prototyping of new EAs, and promote creativeness by making most everything explicit. Moreover, with minimal code change, the proposed framework also includes tools that allow easy parallelism for distributing the most computationally intensive parts of algorithms over a computer cluster. The distribution model brought forward has no hierarchy, enabling each worker to create new tasks and share the total workload of the application. The framework has been proven easy to use even for non trivial algorithms. Furthermore, after only two years of existence, DEAP is already used by several researchers from different domains, studying bloat control in genetic programming to sensor network placement using genetic algorithms. DEAP is an open source project, freely available at `http://deap.googlecode.com`.

Acknowledgements

The DEAP team acknowledges the financial support of the FQRNT (Québec) and NSERC (Canada) and access to the supercomputing facilities of Calcul/Compute Québec/Canada.

7. REFERENCES

[1] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

[2] E. Cantú-Paz. *Efficient and accurate parallel genetic algorithms*. Kluwer, 2000.

[3] P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. Take it EASEA. In *Proc. of Parallel Problems Solving from Nature (PPSN)*, 2000.

[4] Y. Collette, N. Hansen, G. Pujol, D. Salazar Aponte, and R. Le Riche. On object-oriented programming of optimizers – Examples in Scilab. In P. Breitkopf and R. F. Coelho, editors, *Multidisciplinary Design Optimization in Computational Mechanics*, chapter 14, pages 527–565. Wiley, 2010.

[5] M. Dubreuil, C. Gagné, and M. Parizeau. Analysis of a master-slave architecture for distributed evolutionary computations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 36(1):229 –235, 2006.

[6] C. Gagné and M. Parizeau. Open BEAGLE, a versatile EC framework, `http://beagle.gel.ulaval.ca`, 2007.

[7] W. D. Hillis. Co-evolving parasites improves simulated evolution as an optimization procedure. In *Proc. of Artificial Life II*, pages 313–324, 1992.

[8] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: A general purpose evolutionary computation library. In *Proc. of Artificial Evolution*, 2002.

[9] S. Luke. ECJ evolutionary computation system, `http://cs.gmu.edu/~eclab/projects/ecj/`, 2010.

[10] D. Roberts and R. E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.