Bio-Inspired Artificial Intelligence

Prof. Giovanni Iacca
giovanni.iacca@unitn.it

Module 12–Lab Exercises

Introduction

Goal. The goal of this lab is to study the application of quality-diversity algorithms, in particular the Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) to various kinds of problems. We will also investigate the parametrization of the algorithm and its effect on the algorithmic performance.

Getting started. Download the file `12.Exercises.zip` from Moodle and unzip it. In this lab we will use the `qdpy`[1] library, as well as the `deap` library seen in the previous lab. All the exercises are based on the examples available in the library[2], with some modifications. Please note that the `qdpy` library contains many other quality-diversity algorithms, such as Novelty Search, and advanced variants of MAP-Elites. However, in this lab we will use for simplicity only the vanilla version of MAP-Elites.

As usual, each exercise has a corresponding `.py` file. To solve the exercises, you will have to open, edit, and run these `.py` files.

The basic version of MAP-Elites is shown in Algorithm 1. In the pseudo-code, $\mathbf{x}$ and $\mathbf{x}'$ are candidate solutions (i.e., n-dimensional vectors defined in the search space $\mathbf{D}$); $\mathbf{b}'$ is a *feature descriptor*, that is a location in a user-defined *discretized* feature space (which can be seen as a *grid* made of *bins*), corresponding to the candidate solution $\mathbf{x}'$, (i.e., an N-dimensional vector of user-defined features that characterize $\mathbf{x}'$, typically with $N < n$); p' is the performance of the candidate solution $\mathbf{x}'$ (i.e., the scalar value returned by the objective function $f(\mathbf{x}')$; $\mathcal{P}$ is a <feature descriptor, performance> map (i.e., an associative table that stores the best performance associated to each feature descriptor encountered by the algorithm); $\mathcal{X}$ is a <feature descriptor, solution> map (i.e., an associative table that stores the best solution associated to each feature descriptor encountered by the algorithm); $\mathcal{P}(\mathbf{b}')$ is the best performance associated to the feature descriptor $\mathbf{b}'$ (it can be empty); $\mathcal{X}(\mathbf{b}')$ is the best solution associated to the feature descriptor $\mathbf{b}'$ (it can be empty).

Following the pseudo-code, the algorithm first creates the two maps $\mathcal{P}$ and $\mathcal{X}$, which are initially empty. Then, a while loop is executed until a given stop criterion is not met (usually, on the maximum number of function evaluations). Each iteration of the loop evaluates a *batch* of solutions. In the first batch, a given number of solutions are randomly sampled—see the randomSolution() function—in the search space $\mathbf{D}$, which are used for initializing the two maps $\mathcal{P}$ and $\mathcal{X}$. Then, starting from the next iteration, solutions are first randomly selected from the

[1] A Quality-Diversity framework for Python 3.6+: `https://gitlab.com/leo.cazenille/qdpy`
[2] `https://gitlab.com/leo.cazenille/qdpy/-/tree/master/examples`

Algorithm 1 MAP-Elites algorithm

```
$\mathcal{P} \leftarrow \emptyset$                                        ▷ <feature descriptor, performance> map
$\mathcal{X} \leftarrow \emptyset$                                        ▷ <feature descriptor, solution> map
$firstBatchInitialized \leftarrow$ False
while stop criterion not met do
    for $i = 1 \ldots batchSize$ do                    ▷ evaluate a batch of solutions at each iteration
        if not $firstBatchInitialized$ then
            $\mathbf{x}' \leftarrow$ randomSolution()                  ▷ first iteration: only random generation
        else
            $\mathbf{x} \leftarrow$ randomSelection($\mathcal{X}$)                ▷ next iterations: random selection...
            $\mathbf{x}' \leftarrow$ randomVariation($\mathbf{x}$)                                 ▷ ...and variation
        end if
        $\mathbf{b}' \leftarrow$ featureDescriptor($\mathbf{x}'$)
        $p' \leftarrow$ performance($\mathbf{x}'$)
        if $\mathcal{P}(\mathbf{b}') = \emptyset \vee \mathcal{P}(\mathbf{b}') > p'$ then    ▷ if empty bin, or new solution is better (minimiz.)
            $\mathcal{P}(\mathbf{b}') \leftarrow p'$
            $\mathcal{X}(\mathbf{b}') \leftarrow \mathbf{x}'$
        end if
    end for
    if not $firstBatchInitialized$ then
        $firstBatchInitialized \leftarrow$ True                          ▷ first batch evaluated
    end if
end while
return $\mathcal{P}$ and $\mathcal{X}$
```

current map $\mathcal{X}$—through the randomSelection() operator—and then perturbed according to the randomVariation() operator. For each new solution $\mathbf{x}'$, the corresponding feature descriptor $\mathbf{b}'$ and performance p' are then evaluated. At this point, the two maps $\mathcal{P}$ and $\mathcal{X}$ are updated: if the performance associated to $\mathbf{b}'$, $\mathcal{P}(\mathbf{b}')$, is empty (which can happen if this is the first time that the algorithm generates a solution with that feature descriptor), or if it contains a value that is worse than the performance p' of the newly generated solution (in Algorithm 1, we assume a minimization problem, therefore we check the condition $\mathcal{P}(\mathbf{b}') > p'$), the new solution $\mathbf{x}'$ and its performance p' are assigned to the elements of the maps corresponding to its feature descriptor $\mathbf{b}'$, namely $\mathcal{P}(\mathbf{b}')$ and $\mathcal{X}(\mathbf{b}')$. Once the loop terminates, the algorithm returns the two maps $\mathcal{P}$ and $\mathcal{X}$, which can be later analyzed for further inspection and post-processing.

It can be immediately noted how simple the algorithm is. With reference to the pseudo-code, in order to apply MAP-Elites to a specific problem the following methods must be defined:

- randomSolution(): returns a randomly generated solution;
- randomSelection($\mathcal{X}$): randomly selects a solution from $\mathcal{X}$;
- randomVariation($\mathbf{x}$): returns a modified copy of $\mathbf{x}$;
- featureDescriptor($\mathbf{x}$): maps a candidate solution $\mathbf{x}$ to its feature descriptor, $\mathbf{b}$;
- performance($\mathbf{x}$): evaluates the objective function of the candidate solution $\mathbf{x}$.

The first three methods are rather standard, i.e., they can be based on general-purpose operators typically used in EAs. However, it is possible to customize them according to the specific need.

For instance, in the first two exercises of this lab we will use uniform random sampling and uniform random selection for the first two operators. For the variation operator, we will use the `RandomSearchMutPolyBounded` operator provided by `qdpy`, which essentially performs uniform random mutations with a saturation on the bounds of the search space. In the third exercise, we will use instead the typical operators of Genetic Programming.

As for what concerns featureDescriptor($\mathbf{x}$) and performance($\mathbf{x}$), these are obviously problem-dependent: the first one, being dependent on how the user defines the features of interest and the corresponding feature space; the latter, being dependent on the specific objective function at hand. In the exercises, we will see different definitions of performances and descriptors.

Exercise 1

In this exercise, we will use MAP-Elites to "illuminate" the feature space of a benchmark function that we have already used in some of the first labs, namely the Rastrigin function[3], which as you may remember is a highly multimodal problem.

For simplicity, we will use as feature descriptor for MAP-Elites the first two variables of the problem. Note however that, in general, the features used in MAP-Elites can be any property (different from the fitness function) of the solutions to the problem at hand.

To start the experiments, from a command prompt go to the `exercise_1` folder and run[4]:

```
$>python exercise_rastrigin.py
```

At the end of the run, the script will generate a series of plots (see Fig. 1 in the current working directory, namely:

- `activityGrid.pdf`: this map indicates, for each bin, how many times that bin has been updated (i.e., its elite has been replaced) during the evolutionary process;
- `evals_contsize.pdf`: this trend indicates the cumulative number of bins filled during the evolutionary process;
- `evals_fitnessmax0.pdf`: this trend is the usual fitness trend that we have seen in the previous labs (note: in this case the fitness has to be minimized);
- `iterations_nbupdated.pdf`: this trend indicates how many bins are updated at each iteration of MAP-Elites;
- `performancesGrid.pdf`: this is the final "illumination" map that shows how the performance of the elites changes depending on the features at hand (brighter color indicates better results—note that the fitness is normalized in [0,1]).

Furthermore, the script will serialize the final version of the map handled by MAP-Elites in a pickle file named `final.p`, that can be deserialized and manipulated for further analysis.

- What kind of considerations can you make regarding the fitness trend (Is the algorithm able to converge to a reasonably low fitness function? How quick is the convergence?), and the activity grid (For instance, are there regions of the feature space that are visited/updated

[3]`https://pythonhosted.org/inspyred/reference.html#inspyred.benchmarks.Rastrigin`

[4]For all the exercises in this lab you may follow the name of the `.py` file with `-seed SEED`, where `SEED` is an integer value which will serve as the seed for the pseudo-random number generator. This will allow you to reproduce your results. Also, please note that in this document `$>` represents your command prompt, do not re-type these symbols.

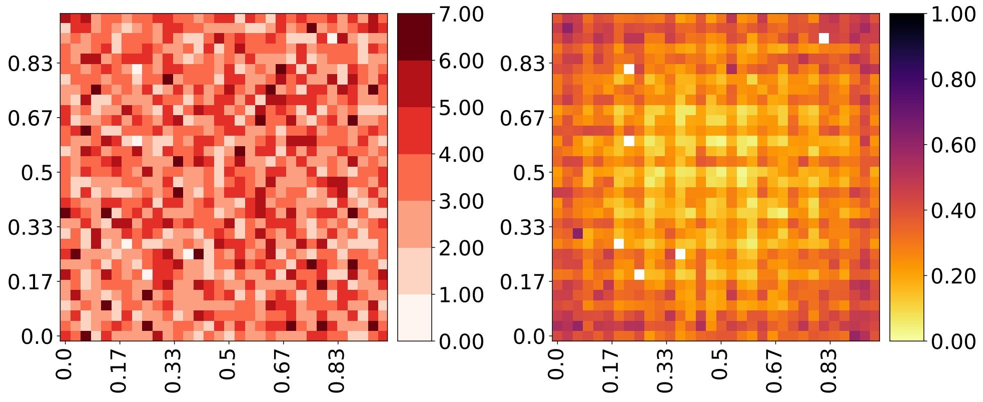

Figure 1: Output of the experiments of the first exercise: activity grid (left); performance grid (right). More plots are available in the exercise folder after the execution of the experiments.

more frequently than others?). What kind of illumination pattern do you observe? Do you see any trend/correlation between performance and features of the map?

- Try to change the parameters of the MAP-Elites algorithm, i.e.,: `NO_BINS`, `MAX_ITEMS_BIN`, `BUDGET`, `BATCH_SIZE`, which indicate, respectively, the number of bins (that is the same for both features), the maximum number of items stored in each bin of the grid, the total budget of the evolutionary process (number of function evaluations), and the batch size, i.e., how many solutions are evaluated at each iteration of MAP-Elites. Focus in particular on `NO_BINS`. What is the effect on the fitness trend and the performance map when you increase or decrease the number of bins?
- Try to change the problem dimension (`PROBLEM_DIM`) to a much larger value, for instance 10 (remember that Rastrigin is a scalable benchmark problem, meaning that it can be defined for any number of variables). Note that in any case the first two variables are taken as features for MAP-Elites. What kind of considerations can you make in this case regarding the illumination pattern and the other aspects (i.e., the fitness trend and the activity grid) of the results? Does illumination become more difficult (i.e., less bins are visited, with poorer performance)? Why?

Exercise 2

This exercise is similar to the previous one. The main difference is that in this case the objective function and feature descriptor are defined by a custom function, see `eval_fn`, that returns for each individual its fitness (`score`) and two features (`fit0` and `fit1`). Note that the fitness and features are based on trigonometric functions and are defined as scalable, i.e., they can be evaluated for any number of variables.

To start the experiments, from a command prompt go to the `exercise_2` folder and run:

```
$>python exercise_custom_eval_fn.py
```

At the end of the run, the script will generate the same plots discussed in the previous exercise, as well as the pickle file containing the raw results.

- What kind of considerations can you make in this case regarding the fitness trend and illumination pattern?
- Also in this case, try to change `NO_BINS` and `PROBLEM_DIM`, and see if you can confirm the observations made in the previous experiment.
- If you want, you could try to change the custom function definition in `eval_fn` and replicate the experiment with a different setting. What kind of results do you obtain?

Exercise 3

In this exercise we will use MAP-Elites in combination with Genetic Programming (as implemented in `deap`, see the exercises from the previous lab) to solve a symbolic regression problem. The problem is exactly the same as the one seen in the first exercise of the previous lab on Genetic Programming (in which the goal was to fit a polynomial function), apart from the fact that in this case we will investigate how MAP-Elites illuminates a feature spaces characterized by the size of the tree (the number of nodes) and its depth. Similarly to the exercise seen in the lab on GP, the fitness in this case is the mean square error calculated on the training points (to be minimized). As both features can be seen as proxy for the complexity of the evolved trees, this experiments can give insights on how the performance of the trees depends on their complexity.

To start the experiments, from a command prompt go to the `exercise_3` folder and run:

```
$>python exercise_deap_map-elites_SR.py
```

Similarly to the previous exercises, at the end of the run the script will generate some plots (in this case only `activityGrid.pdf` and `performancesGrid.pdf`) as well as the pickle file containing the raw results. Also, note that the structure of the best tree is displayed on the terminal (in Reverse Polish notation).

- Is the algorithm able to approximate the given polynomial? If not, try to change some parameters of the algorithm and see if you can improve the results. Note that in this case there is an additional parameter (`INIT_BATCH_SIZE`), that is the size of the first batch (used to initialize the map). This is usually set to be bigger than the batch size at the subsequent iterations of the algorithm.
- Try to change the generator function (e.g., to include trigonometric functions) defined in the method `generatorFunction`. Is the algorithm able to approximate more complicated generator functions? Which parameters can you change to improve the results?
- What kind of illumination pattern do you observe in the various trials? Do you see any trend/correlation between performance and features (i.e., the size and depth of the tree)?

Instructions and questions

Concisely note down your observations from the previous exercises (follow the bullet points) and think about the following questions.

- Do you think there is a trade-off between quality and diversity, or one aspect is more important than the other? If so, which one, and in which circumstances?
- In which kind of applications do you think that MAP-Elites (and quality-diversity algorithms in general) could be useful? Why?