Bio-Inspired Artificial Intelligence

Prof. Giovanni Iacca
giovanni.iacca@unitn.it

Module 10–Lab Exercises

Introduction

Goal. The goal of this lab is to investigate two examples of Competitive Co-Evolution: the first one dealing with a robotic prey-predator experiment, the second one dealing with a computational model for sorting algorithms, named Sorting Network.

Getting started. Download the file `10.Exercises.zip` from Moodle and unzip it. This lab continues the use of the *inspyred* framework for the Python programming language seen in the previous labs. If you did not participate in the previous labs, you may want to look those over first and then start this lab's exercises. Additionally, in this lab we will use a custom 2-D robot Java robotic simulator (**for more details, see module 9's exercises**), and another Python library for Evolutionary Computation named *deap*[1]. With respect to *inspyred*, *deap* has some nice features such as a simple template for co-evolutionary algorithms, and an easy-to-use Genetic Programming implementation. We will see the latter in the next lab.

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

Note once again that, unless otherwise specified, in this module's exercises we will use real-valued genotypes and that the aim of the algorithms will be to *minimize* the fitness function $f(\mathbf{x})$, i.e. lower values correspond to a better fitness!

Exercise 1

In this exercise we will replicate the prey-predator competitive co-evolution experiments we have seen during the lecture. To do so, we will use a custom 2-D robot Java robotic simulator. In this case by default the simulator generates an "empty" (without obstacles) arena 2m×2m, with two E-puck robots that at the beginning of the simulation are placed at opposite corners of the arena. One robot has the role of a "prey", the other acts as a "predator". The goal of the prey is to avoid being "captured" (i.e., get in contact with) the predator, whereas the goal of the predator is to "capture" (get in touch with) the prey. Figure 1 shows some examples of evolved robot behaviors.

[1] Distributed Evolutionary Algorithms in Python: `https://github.com/DEAP/deap`

By default, both robots are controlled by a Feed Forward Neural Network (FFNN) with 8 IR sensor inputs to detect walls, 2 inputs for distance and bearing towards the "target" -in this case towards the other robot-, bias, 5 hidden nodes, and 2 output nodes for controlling the left and right wheels of the robot (**for more details, see module 9's exercises**). All nodes use a sigmoid activation function, with inputs/outputs normalized in $[0, 1]$ and weights evolved in the range $[-3, 3]$. Once again, all the details of the simulator can be defined by means of a configuration file, see the file `Java2dRobotSim/config/config2d2_prey_predator.txt`.

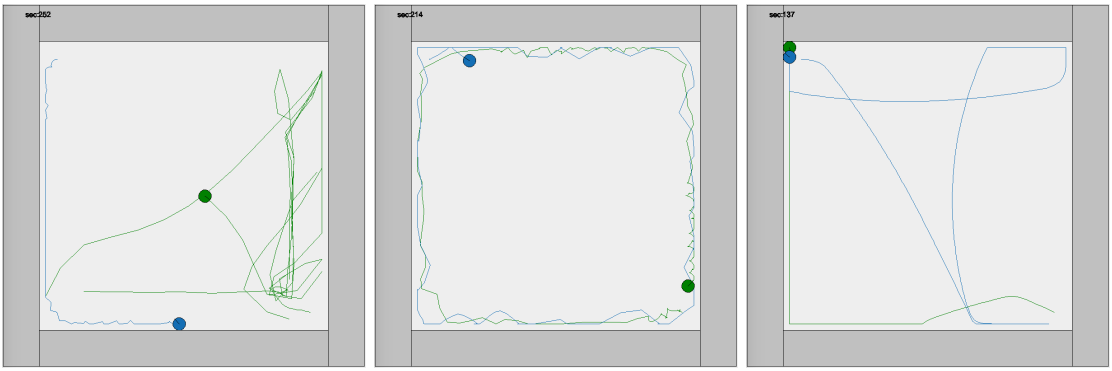

Figure 1: Some examples of evolved prey-predator strategies (prey is green, predator is blue).

In order to co-evolve the robot behaviors, two separate populations of preys and predators are evolved *synchronously* by two Evolutionary Algorithms that run in separate *alternating* threads (i.e., the evaluation part of the algorithm alternates between the two threads: evaluate preys, evaluate predators, evaluate, preys, ...). Both algorithms can be parametrized differently (but, by default they use the same parameters), and are configured to keep an archive of best solutions (i.e., FFNN controllers) found during the evolution.

At each generation, each algorithm simulates its own solutions in the current population against (a subset of) the solutions taken from the archive kept by the other algorithm. For efficiency purposes, the simulation takes as input a list of `numRobots` candidate solutions that is split in two halves, the first one containing preys, the second one containing predators. Then the simulator lets each i-th robot (a prey), for i in [0,`numRobots`/2), "compete" against the (i+`numRobots`/2)-th robot (the corresponding predator). Therefore each prey and each predator can be repeated in the list multiple times, in order to generate all the needed pairwise competitions between preys and predators. Before starting the experiments, spend some time to have look at the script `exercise_prey_predator.py` and understand its main steps (in particular, see the method `evaluator` of the class `RobotEvaluator`).

To start the experiments, from a command prompt run[2]:

```
$>python exercise_prey_predator.py
```

Depending on how fitness is defined for both preys and predators (once again, see the method `evaluator` of the class `RobotEvaluator`), different robot behaviors can be obtained. For each robot, the simulator returns three main quantities that can be used/combined differently to drive the co-evolution in different ways, namely:

[2]For this exercise you may follow the name of the `.py` file with 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.

1. `finalDistanceToTarget`: the final (measured at the end of the simulation) distance to the "target" robot.
2. `minDistanceToTarget`: the minimum (measured during the simulation) distance to the "target" robot.
3. `timeToContact`: the time to contact (in timesteps, in the range [0,`nrTimeStepsGen`]).

By default, the preys are evolved to *maximize* their `minDistanceToTarget`, while the predators are evolved to *minimize* it. Please note that the two distance metrics range in $[0, d_{max}]$ (in mm) where d_{max} is pre-computed as the maximum distance in the given environment (length of the diagonal of the arena). Also, note that, due to the aforementioned structure of the list of candidate solutions taken as input, preys are kept in the first `numRobots`/2 elements of the list, while predators are kept in the remaining elements, such that different fitness functions can be used for preys and predators.

Furthermore, the way the two algorithms update the corresponding archives of best solutions can be controlled by the following parameters:

1. `numOpponents`: the number of opponents against which each robot competes at each generation (default: 1).
2. `archiveType`: the way competition with individuals from the archive is performed; possible values are {GENERATION, HALLOFFAME, BEST} (default: BEST). When GENERATION is selected, generational competition is performed, i.e. each algorithm keeps an archive containing one best solution for each of the previous `numOpponents` generations, such that at each generation each robot competes against the best opponents from those `numOpponents` generations (Master Tournaments). Similarly, when HALLOFFAME is selected each algorithm keeps an archive containing one best solution for each of the previous generations, however in this case there is no limit on the number of solutions kept in the archive (whose size increases along generations), and `numOpponents` indicates the number of solutions which are randomly sampled from the archive to perform pairwise competitions (see the lecture slides). When BEST is selected, a greedy approach is taken: in this case indeed each algorithm keeps in the archive the best `numOpponents` opponents from *all* previous generations (not necessarily one per generation), and each robot competes against those opponents.
3. `archiveUpdate`: the way fitness is aggregated for each robot; possible values are {WORST, AVERAGE} (default: WORST). When WORST is selected, each robot competes against `numOpponents` opponents from the other algorithm's archive and its final fitness is set to its worst value obtained across `numOpponents` competitions (worst-case scenario). When AVERAGE is selected, the final fitness of each robot is set to its average value obtained across `numOpponents` competitions.
4. `updateBothArchives`: the way archives are updated at each generation; possible values are {True, False} (default: False). When False is selected, each algorithm updates only its own archive. When True is selected, each algorithm also recomputes the fitness of the opponents and updates the other algorithm's archive if needed (this is a non-standard feature).

You can "replay" each generation by running this command:

```
$>python run_candidates.py config2d2_prey_predator.txt
 SEED/candidates_NUMGEN_ROBOTTYPE.txt
```

where `NUMGEN` is the generation number and `ROBOTTYPE` is either `Predators` or `Preys`. Note that in this case, depending on the selected type of robot, you will see multiple pairwise prey-predator competitions as if they were executed independently from each other (i.e. the robot interaction is only one-to-one instead of many-to-many).

You can also "replay" just the best candidates (at the last generation) by running this command:

```
$>python run_candidates.py config2d2_prey_predator.txt
 SEED/best_candidates.txt
```

where `SEED` is the seed used in the simulation (e.g., 1542609572).

Finally, you can plot the fitness trends of a specific run (whose results are stored in the folder `SEED`) by running this command:

```
$>python plot_run.py SEED
```

Consider the following experiments:

- Try out different parameter combinations of `numOpponents`, `archiveType`, `archiveUpdate`, and `updateBothArchives`, and observe what kind of robot behavior is evolved. Can you find cases where the prey "wins"? Can you find cases where the predator "wins"?
- Try to change the fitness formulation and observe what kind of behavior is evolved. Remember to change the two flags `problemPreysMaximize` and `problemPredatorsMaximize` properly, according to the way you defined the fitness function.
- (Optional) Try to change the EA's and FFNN's parameters to see if/how results change depending on those values.

Exercise 2

In this exercise we will use a competitive co-evolutionary approach for evolving a *Sorting Network* (SN)[3]. A SN is an abstract mathematical model of a network of *wires* and comparator modules (*connectors*) that is used to sort a sequence of numbers fed as input to the network. Each comparator connects two wires and sorts the values by outputting the smaller value to one wire, and the larger value to the other. For further details on the theory behind SNs, refer to the corresponding Wikipedia page. For illustration purposes, an example of SN is shown in Figure 2.

The code of this exercise is based on one of the *deap* examples and provides a nice template for a generic competitive co-evolutionary algorithm in this framework. Similarly to the previous exercise, also in this case two separate populations are kept, one for *hosts* (where each host is a candidate sorting network) and one for *parasites* (where each parasite is an array of a given number of shuffled sequences to be sorted). For simplicity, we consider here a binary sorting problem, where input sequences are (unsorted) *binary strings* of fixed size, such as {1,0,0,0,1,1,0,1}. The size of the evolved SNs (also called *depth*, i.e. its number of connectors) is instead variable and evolves with the hosts[4]. The goal of a host is to sort all sequences in the competing parasite,

[3] `https://en.wikipedia.org/wiki/Sorting_network`

[4] Note that the depth of a SN is a measure of its algorithmic complexity. Optimal (minimum-depth) SNs are currently known only up to a inputs sequences of size equal to 17, see the corresponding Wikipedia page. In principle, one could use EAs also for minimizing the depth, together with the sorting errors, either based on a single objective (with scalarization) or on a multi-objective approach optimizing depth and sorting errors

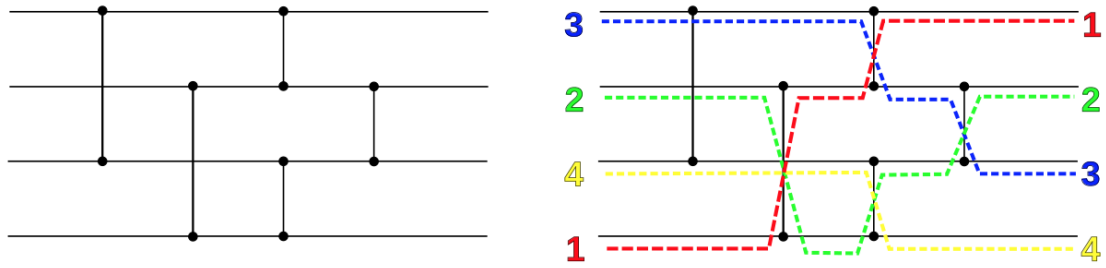

Figure 2: An example of Sorting Network, and its functioning with an input sequence {3,2,4,1}.

while the goal of a parasite is to induce errors in the competing host. The fitness of a host is therefore calculated as the total number of sequences that it could not sort properly, from those sequences present in the parasite against which that host competed. The fitness of the parasite is exactly the same value. Obviously, the fitness of hosts (number of sorting errors) must be *minimized*, while the fitness of parasites should be *maximized*.

While the *deap* library is based on some different concepts and implementation details with respect to the *inspyred* library we have used so far, its working principles are quite straightforward and can be understood rather easily. Take some time to have a look at the source code in the file `exercise_sortingnetwork.py` (note however that the implementation of the Sorting Network is available in the module `sortingnetwork.py`). The relevant parameters of the Evolutionary Algorithm, hosts and parasites, can be found at the beginning of the script. In particular, consider the parameters `INPUTS`, `POP_SIZE_HOSTS`, `POP_SIZE_PARASITES`, `HOF_SIZE`, `MAXGEN`, `H_CXPB`, `H_MUTPB`, `P_CXPB`, `P_MUTPB`, `H_TRNMT_SIZE`, `P_TRNMT_SIZE`, `P_NUM_SEQ`. Note that in this case (differently from the previous exercise) the two populations are evolved within a single thread, and that at each generation *all hosts in the current population* are tested against *all parasites in the current population*. A Hall-of-Fame is kept to store the SNs displaying the best performance across generations, and updated whenever a new SN has a better performance (smaller number of sorting errors) than the worst SN in the Hall-of-Fame. The final output of the script is a graphical representation of the best SN in the Hall-of-Fame, and the number of sorting errors it suffers on *all possible input sequences* of fixed input size equal to `INPUTS`. Also, the usual plot with the min/max/avg fitness trends is provided.

Run the script `exercise_sortingnetwork.py` to perform the competitive co-evolutionary experiment. Also in this case you can pass as argument to the script a specific seed.

- Is the co-evolutionary algorithm able to evolve an optimal (without sorting errors) SN, in the default configuration?
- Try to investigate this problem in different configurations. In particular, focus on the effect of the size of the input sequences (`INPUTS`), the number of input sequences per parasite (`P_NUM_SEQ`), and the two population sizes (`POP_SIZE_HOSTS` and `POP_SIZE_PARASITES`). If needed, also change the size of the Hall-of-Fame (`HOF_SIZE`) and the number of generations (`MAXGEN`). What conclusions can you draw? For instance: What makes the problem harder? What is the effect of `P_NUM_SEQ`? What can you do to solve the harder problem instances?

separately.

Instructions and questions

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

- Can you provide some example applications where you think a competitive co-evolution approach could be used?
- Can you think of some other competitive co-evolutionary dynamics in nature different from the prey-predator case?