Bio-Inspired Artificial Intelligence

Prof. Giovanni Iacca
giovanni.iacca@unitn.it

Module 6–Lab Exercises

Introduction

Goal. The goal of this lab is to familiarize yourself with Particle Swarm Optimization and study the effect of parametrization on the algorithmic performance.

Getting started. Download the file `06.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.

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

As a first exercise, we will run a simple 2D Boids simulator, based on the Reynolds' flocking rules we have seen during the lectures[1]. Although this exercise is not strictly related to PSO, it provides a good source of inspiration (and intuition) on how PSO works.

To run the simulator, from the `src` folder run `java -jar Boids.jar`. As shown in Figure 1, the simulator allows you to change various aspects of the simulation, specifically the total number of boids, the number of neighbors whose information is collected by each boid (to determine cohesion, alignment, and separation), and the relative weights of each of the 3 flocking rules (behavior coefficients). Spend some time with the simulator, and try different simulation configurations.

To help you figure out the behavior of the boids, you can find below the implementation of the `Boid` class extracted from the source code of the simulator. In particular, check the `updateVelocity` method and the methods called in it.

[1]Source code taken from `https://github.com/k5md/Boids`.

```java
class Bird {
  Vector position;

  Vector velocity;

  public Bird(Vector position, Vector velocity) {
    this.position = position;
    this.velocity = velocity;
  }

  public void updateVelocity(Bird[] birds, int xMax, int yMax, double
      cohesionCoefficient, int alignmentCoefficient, double
      separationCoefficient) {
    this.velocity = this.velocity.plus(cohesion(birds, cohesionCoefficient))
      .plus(alignment(birds, alignmentCoefficient))
      .plus(separation(birds, separationCoefficient))
      .plus(boundPosition(xMax, yMax));
    limitVelocity();
  }

  public void updatePosition() {
    this.position = this.position.plus(this.velocity);
  }

  public Vector cohesion(Bird[] birds, double cohesionCoefficient) {
    Vector pcJ = new Vector(new double[] { 0.0D, 0.0D });
    int length = birds.length;
    for (int i = 0; i < length; i++)
      pcJ = pcJ.plus((birds[i]).position);
    pcJ = pcJ.div(length);
    return pcJ.minus(this.position).div(cohesionCoefficient);
  }

  public Vector alignment(Bird[] birds, int alignmentCoefficient) {
    Vector pvJ = new Vector(new double[] { 0.0D, 0.0D });
    int length = birds.length;
    for (int i = 0; i < length; i++)
      pvJ = pvJ.plus((birds[i]).velocity);
    pvJ = pvJ.div(length);
    return pvJ.minus(this.velocity).div(alignmentCoefficient);
  }

  public Vector separation(Bird[] birds, double separationCoefficient) {
    Vector c = new Vector(new double[] { 0.0D, 0.0D });
    int length = birds.length;
    for (int i = 0; i < length; i++) {
      if ((birds[i]).position.minus(this.position).magnitude() <
          separationCoefficient)
        c = c.minus((birds[i]).position.minus(this.position));
    }
    return c;
  }

  public Vector boundPosition(int xMax, int yMax) {
    int x = 0;
    int y = 0;
    if (this.position.data[0] < 0.0D) {
      x = 10;
    } else if (this.position.data[0] > xMax) {
      x = -10;
    }
    if (this.position.data[1] < 0.0D) {
      y = 10;
    } else if (this.position.data[1] > yMax) {
      y = -10;
    }
    return new Vector(new double[] { x, y });
  }

  public void limitVelocity() {
    int vlim = 100;
    if (this.velocity.magnitude() > vlim) {
      this.velocity = this.velocity.div(this.velocity.magnitude());
      this.velocity = this.velocity.times(vlim);
    }
  }

  public String toString() {
    return new String("Position: " + this.position + " Velocity: " + this.
        velocity);
  }
}
```

- What is the effect of each behavior coefficient?
- Which combination of coefficients leads to the most “natural” flock behavior?

Exercise 2

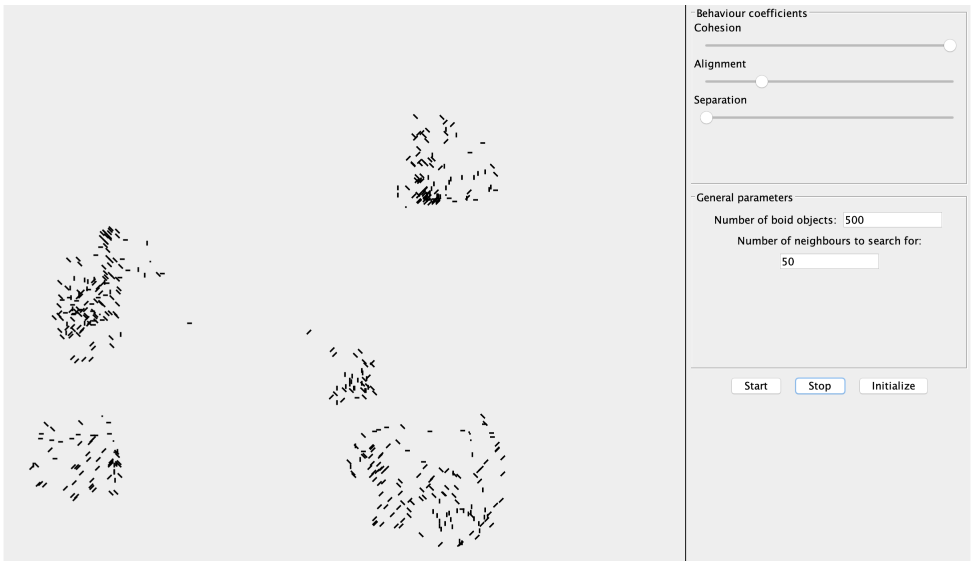
In this exercise we will perform a comparative analysis of the results of Genetic Algorithm (as seen in Lab 2), Evolution Strategies (as seen in Lab 3) and Particle Swarm Optimization. To start the experiments, from a command prompt run[2]:

```
$>python exercise_2.py
```

The script will perform a single run of GA, ES and PSO, on one of the benchmark functions we have seen in the previous labs. As usual, the algorithm parametrization is shown in the code and can be easily modified.

- What kind of behavior does PSO have on different benchmark functions (change the parameter `args["problem_class"]` to try at least a couple of functions), in comparison with the EAs? Does it show better or worse results? Does it converge faster or not?

[2]For all the exercises in this lab 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.



Number of boid objects: 500

Number of neighbours to search for: 50

Figure 1: A screenshot of the 2D Boids simulator.

- Increase the problem dimensionality (`num_vars`, by default set to 2), e.g. to 10 or more. What do you observe in this case?
- Change the population size (by changing `args["pop_size"]`, by default set to 50) and the number of generations (by changing `args["max_generations"]`, by default set to 100), such that their product is fixed (e.g. 50×100, 100×100, etc.). Try two or three different combinations and observe the behavior of the three different algorithms. What do you observe in this case? Is it better to have smaller/larger populations or a smaller/larger number of generations? Why?

Exercise 3 (optional)

Now that you have some intuition on how an existing implementation of PSO works (as well as the underlying biological inspiration), you might find useful implementing a PSO algorithm on your own (in Python or any other language of your choice). As we have seen in the lecture, this can be coded in just a few lines. If you want, try to implement a **simple** version of PSO (with the parametrization found in the previous exercise), to minimize for instance the Sphere function. You could also try to include some constraints and a constraint handling technique.

Instructions and questions

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

- When do you think it is useful to have a lower (higher) cognitive learning rate? What about the social learning rate?

- From a biological point of view, which neighborhood topology do you consider as the most plausible?