Bio-Inspired Artificial Intelligence

Prof. Giovanni Iacca
giovanni.iacca@unitn.it

Week 7 Lab Exercises

Introduction

Goal. The goal of this lab is to study two of the most famous discrete discrete optimization benchmark problems - the Traveling Salesman Problem (TSP) and the Knapsack problem. Discrete optimization problems often present difficulties for naïve Evolutionary Computation approaches, where special care must be taken to generate and maintain feasible solutions and/or to sufficiently penalize infeasible solutions. Here we will see how Ant Colony Optimization can solve this kind of discrete problems much more efficiently than Evolutionary Computation approaches. In addition to that, we will study the effect of parametrization on the performance of Ant Colony Optimization.

Getting started. Download the file `07.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 week'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

Traveling Salesman Problem In the Traveling Salesman Problem (TSP)[1], a salesperson is expected to make a round trip visiting a set of N cities, returning the starting city. The requirement is to *minimize* the total distance traveled. In the simplest version of the TSP, that we consider in this exercise, it is assumed that it is possible to travel from any city i to any city j (i.e., cities form a fully-connected graph), and the distance is the same both ways.

To specify a TSP instance, we need the spatial coordinates (x, y) for each of the N cities, and an adjacency matrix ($N \times N$) describing the pairwise distances between cities. These can be

[1]See `https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html`

computed as Euclidean distances between cities, or in general can be specified according to a given distance function (which may take into account travel cost, time, and other factors).

Candidate solutions for the TSP can be easily represented as *permutations* of the list of city indices (enumerating the order in which cities should be visited). For instance, if there are 5 cities, then a candidate solution might be [4, 1, 0, 2, 3]. In case of Evolutionary Computation approaches, a simple sequence representation can be used, coupled with mechanisms that ensure that the candidate solutions remain feasible during crossover and mutation. This can be accomplished by *ad hoc* genetic operators (which in the *inspyred* framework are dubbed as "variators"), namely the *partially matched crossover* and the *inversion mutation*, as shown `exercise_1.py`. As an alternative, we can use Ant Colony Optimization that is naturally suited for solving graph-like problems such as the TSP and, differently from Evolutionary Computation approaches, does not require any specific operator.

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

```
$>python exercise_1.py
```

The script will perform a single run of Ant Colony Systems (ACS) and a customized Evolutionary Algorithm (EA) and show you the fitness trends of both algorithms, as well as a map corresponding to the best solutions provided by the two algorithms. Note that for both algorithms the fitness function is defined as the reciprocal of the total distance traveled ($f(x) = 1/\sum d$), to be *maximized*, such that maximizing $f(x)$ corresponds to minimizing the total distance.

- Execute the script on different problem instances (this can be obtained by changing the variable `instance` in `exercise_1.py`) and observe the behavior of the two algorithms on each instance[3]. **Note that the run time can be quite long (several seconds) on larger instances** (especially on `att48`). Which algorithm provides the best solution in most cases? What can you say about the number of function evaluations needed to converge?

Exercise 2

Knapsack problem In the Knapsack problem[4], we are given a knapsack of fixed capacity C. We are also given a list of N items, each having a weight w and a value v. We can put any subset of the items into the knapsack, as long as the *total weight* of the selected items does not exceed C. In the most general formulation of the problem, it is possible to select the same item multiple times. We aim to *maximize* the *total value* of the selection, which is the sum of the values of each item we put into the knapsack. Thus, a solution of the Knapsack problem is a subset S of the N items, each considered a certain number of times, for which the total weight is less than or equal to C, and which maximizes the total value. If the same item can be selected multiple times, the problem is called "Knapsack with duplicates", otherwise if any item can be selected only once, the problem is called "0/1 Knapsack".

[2]For all 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.

[3]Note that you can also create your own instances by editing the two variables `points` -a set of coordinates (x, y)- and `distances` -an adjacency matrix that can be computed e.g. by Euclidean distances between points.

[4]See `https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html`

Candidate solutions for the Knapsack problem can be represented as either a binary list (for the 0/1 Knapsack) or as a list of non-negative integers (for the Knapsack with duplicates). In each case, the length of the list is the same as the number of items, and each element of the list corresponds to the quantity of the corresponding item to place in the knapsack. In case of Evolutionary Computation approaches, we can use as genetic operators *uniform crossover* and *Gaussian mutation*, as shown `exercise_2.py`. The reason we can use Gaussian mutation, even though the candidates are composed of discrete values, is because the bounder created by the Knapsack benchmark is an instance of `DiscreteBounder`, which automatically moves an illegal component to its nearest legal value (an integer, in this case). Once again, as an alternative we can use Ant Colony Optimization.

In this exercise, we will focus on the **0/1 Knapsack problem** (this is obtained by setting `duplicates=False` in `exercise_2.py`). To start the experiments, from a command prompt run `exercise_2.py`. The script will perform a single run of Ant Colony Systems (ACS) and a customized Evolutionary Algorithm (EA) and show you the fitness trends of both algorithms.

- Execute the script on different problem instances (this can be obtained by changing the variable `instance` in `exercise_2.py`) and observe the behavior of the two algorithms on each instance[5]. **Note that the run time can be quite long (several seconds) on larger instances**. Which algorithm provides the best solution in most cases? What can you say about the number of function evaluations needed to converge?

Exercise 3

In this exercise we will focus on the **Knapsack with duplicates** (this is obtained by setting `duplicates=True` in `exercise_3.py`). To start the experiments, from a command prompt run `exercise_3.py`. Once again, the script will perform a single run of Ant Colony Systems (ACS) and a customized Evolutionary Algorithm (EA) and show you the fitness trends of both algorithms.

- Execute the script on different problem instances (this can be obtained by changing the variable `instance` in `exercise_3.py`) and observe the behavior of the two algorithms on each instance[6]. **Note that the run time can be quite long (several seconds) on larger instances**. Which algorithm provides the best solution in most cases? What can you say about the number of function evaluations needed to converge?
- Do you observe any difference on the algorithmic behavior between this exercise and the previous one?

Instructions and questions

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

[5] Note that you can also create your own instances by editing the two variables `items` -a set of pairs $< w, v >$- and `capacity`. Both variables must be expressed as integer values.

[6] Also in this case you can also create your own instances by editing the two variables `items` and `capacity`.

- What are the main differences between continuous and discrete optimization problems? Do you think that any of these two classes of problems is more difficult than the other?
- Why is ACS particularly suited for discrete optimization?
- Consider the two versions of the Knapsack problem (0/1, and with duplicates). Which of the two problems is more challenging from an optimization point of view? Why?