Papareto: an object-oriented self-adaptive framework for evolutionary computing in Java

January 8, 2014

1 Introduction

1.1 A few words about evolutionary computing

In computer science, evolutionary computation is a subfield of artificial intelligence (more particularly computational intelligence) that involves combinatorial optimization problems.

Evolutionary computation uses iterative progress, such as growth or development in a population. This population is then selected in a guided random search using parallel processing to achieve the desired end. Such processes are often inspired by biological mechanisms of evolution.

Evolutionary algorithms form a subset of evolutionary computation in that they generally only involve techniques implementing mechanisms inspired by biological evolution such as reproduction, mutation, recombination, natural selection and survival of the fittest. Candidate solutions to the optimization problem play the role of individuals in a population, and the cost function determines the environment within which the solutions "live" (see also fitness function). Evolution of the population then takes place after the repeated application of the above operators.

In this process, there are two main forces that form the basis of evolutionary systems: Recombination and mutation create the necessary diversity and thereby facilitate novelty, while selection acts as a force increasing quality.

Many aspects of such an evolutionary process are stochastic. Changed pieces of information due to recombination and mutation are randomly chosen.

1.2 General description

Papareto is a object-oriented Java framework for the development of evolutionary solutions to computational problems.

Other Java frameworks for evoluionary computing include ECJ, WatchMaker, JGAP, etc. *Papareto* differs from these frameworks in a number of ways.

First, unlike its competitors, *Papareto* does not fall into the category of genetic frameworks because it does not consider genetic representations (chromosomes) of individuals. Individuals are actually not encoded at all since *Papareto* directly deals with application-level objects. Doing this has two advantages:

- it avoids the high cost of encoding/decoding;
- it allows the specification of potentially more meaningful application-specific operators, thereby limiting the creation of non-viable individuals.

Second, *Papareto* is self-adaptive in the two following ways:

- it comes with a multi-threaded parallel execution model that dynamically adapts the number of threads in accordance to the constantly evolving load of the computer;
- all along the evolution process, *Papareto* self-evaluates the performance of the evolutionary operators in use in order to benefit at best of most efficient ones.

Third, *Papareto* does not aim at implementing all known evolutionary strategies or execution techniques for them, instead it keeps it as simple as possible, focusing on accessibility for the Researchers and Engineers who are using it, as well as on performance, brought by hereinbefore mentionned mechanisms.

Fourth, the object-oriented API of *Papareto* does not expose the technical concept of evolutionary algorithm. Instead its defines the natural concept of a *population* which evolves along *generations* of individuals.

The primary objective for developing an (ad hoc) evoluationary framework was to give the *Grph* library the ability to generate particular graph instances. Once done, the code was extracted from the source code of *Grph* and was made available as a separate project called *Papareto*.

2 Algorithm

The evolutionary algorithm behind *Papareto* works as follows.

A population is a set of n individuals. n is called the *size of the population*. To each individual i is associated its fitness $f(i) \in]+\infty + \infty[$.

The population evolves in an iterative fashion. Every iteration aims at building up a new generation of individuals which have greater fitness.

An iteration of the algorithm consists in:

expanding the population of o new individuals. o is the size of teh offspring of the population. More precisely, until the size of the population reaches $s + o$, the algorithm:

1. use binary tournament to choose two individuals i_1 and i_2 among the individuals in the population; it is possible that $i_1 = i_2$;
2. use a crossover operator to create a new individual c out of i_1 and i_2;

3. with a given probability, use a mutation operator to alter (mutate) the new individual;
4. adds the new individual to the population.

retains in the population only the s individuals with greatest fitness and discarding all the others.

Unless the user wants a specific evolution strategy, the algorithm iterates until two-subsequent generations exhibit no improvement of the fitness.

2.1 Parameters

The behavior of the evolutionary algorithm can be altered by the following parameters. It is important to note that an adequate value for a given operator is application-dependent. There are unfortunately not rules which apply to all problems.

2.1.1 Size of the population

By default, the size of the population is set to $s = 100$, and its offspring size is set to $o = 100$. These parameters can be modified even along the runtime of the algorithm.

The bigger, the more the population can have variety.

2.1.2 Offspring size

Defines how big the population will be after a new generation is created, and before best individuals are selected.

2.1.3 Allow duplicates

For example, consider a population of strings. The crossover appends the prefix of one parent to the suffix of the other parent. The resulting child is then mutated by the removal of one random character.

$$(ab, th) \xrightarrow{crossover} ah \xrightarrow{mutation} a$$

$$(tf, da) \xrightarrow{crossover} ta \xrightarrow{mutation} a$$

This example illustrates the situation in which distincts parents may entail the generation of the same child.

2.1.4 Asynchronous updates of the population

When using asynchronous updates of the population, as soon as a new child is created, it is added to the population. This allows this new individual to act as a parent for another new individual in the same generation.

Doing this makes the algorithm converge faster, but not necessarily to the global optimum.

By default, if the asynchronous updates are enabled, all new individuals are added to the population as soon as they are instantiated. It is possible to select for each given individual if it participates or not to asynchronous updates. This is done by overriding the method:

```
1 @Override
2 protected boolean participateToAsynchronousUpdating(Individual<E> i)
3 {
4         double bestFitness = get(0).fitness;
5         double worstFitness = get(size() - 1).fitness;
6         double avgFitness = (bestFitness + worstFitness) / 2;
7         return i.fitness < avgFitness;
8 }
```

The individual that are not selected to participating to asynchronous updates will be added to the population after all new individuals are created.

3 Self-adaptive parallelisation

The expansion of the population is a very costly process since it consists in:

- the creation of new individuals, which are potentially large objects;
- the computation of the fitness for all new individuals.

All processes of "creating then evaluating" a new individual are independant to each other. Consequently they can be executed in parallel with no need for synchronization whatsoever.

Before *Papareto* makes a new generation, it senses the current load of the computer and computes the number c of free cores. It then creates $4 * c$ threads that will compute in parallel the new generation.

The advantage is this adaptive strategy is to take maximum advantage of the computer's evolving computational resources and, at the same time, to prevent the system to overload.

4 Multi-objective optimization

4.1 Defining objectives

Objectives are created by declaring methods named `computeFitness`*xxx*`()`.

4.2 Comparing individuals

Two individuals are considered equal if

5 Monitoring

Papareto comes with a basic graphical interface to the evolution of a given population. This interface enables the user to real-time monitor the improvement of the fitness, the effectiveness of the crossover and mutation operators, the performance of the algorithm, etc. The following line of code activates the monitoring graphical interface:

```
1 Population p = ...
2 p.monitor();
```

For the moment, this interface does not enable any interaction with the running evolutionary algorithm, instead it only propose observation features.