Chapter 1.14
Genetic Programming

Pierre Collet
Université du Littoral Côte d'Opale, France

INTRODUCTION

Genetic programming (GP) is still rather unknown, even though it has recently obtained spectacular results: John Koza showed in his latest book (Koza et al., 2003) that genetic programming can routinely produce solutions that are competitive with human intelligence, without requiring one to be an expert in the domain of the problem to be solved.

A Bit of History

The idea of evolving computer programs dates back to the dawn of computing. Back in 1958, Friedberg made several attempts to have a computer program itself (Friedberg, 1958; Friedberg, Dunham, & North, 1958) using what would now be called mutations. He started with a "population" of random programs, and modified the contents stochastically, trying to improve the results.

Later on, Smith (1980), who was working on learning classifier systems, introduced small programs in the rules he was evolving. However, the modern vision of genetic programming starts with a small but seminal paper by Cramer (1985), who uses a tree-like variable size structure to represent a program. Programs are not written in LISP (as suggested by Koza later on) but in TB (a tree version of the JB language). Along with mutation, Cramer also uses a standard subtree crossover, introduces as well a mono-parental crossover, and insists on the necessity to create closed genetic operators. Above all, he evolves his population of programs with an evolutionary engine.

All the seeds were therefore present for the domain to grow, but as for Manderick and Moyson and Ant Colony Optimization (cf. Chapter III), Cramer somehow failed to promote his work enough, and nothing major happened in this domain for several years. In fact, another major problem was that computers of this era were not powerful enough to obtain really good results with GP.

Genetic Programming à la Koza

By the beginning of the 1990s, genetic programming made its comeback thanks to Koza (1989, 1992, 1994; Koza et al., 1999, 2003) who put in a lot of energy (and computer power) to develop the ideas introduced by Cramer and his predecessors.

Genetic programming is nothing else than standard evolutionary techniques (described in Chapter IV) applied to individuals that implement programs. Standard evolutionary algorithms evolve potential solutions to a problem to be optimized. Most of the time, EA individuals are made of a list of parameters that are passed over to a "fitness function" used to evaluate the individual.

In genetic programming, an individual is a program, or more often a function. The main (and only?) difference with EAs is that GP *executes the individuals to evaluate them.* Another difference is that in most of the cases, GP uses variable length individuals where standard EAs use fixed size individuals.

This chapter presents standard genetic programming *à la* Koza, including hints, suggestions, and pointers to state-of-the-art papers that will hopefully allow newcomers to obtain good results with this delicate technique.

STANDARD GENETIC PROGRAMMING

Representation of an Individual

As is the case in EAs, using the good representation for a particular problem is quite essential, because the chosen representation more or less determines the search space in which the individuals will evolve: it will be very difficult to obtain an iterative program with a representation that does not allow loops or recursive calls. On the contrary, if everything is allowed, the search space may be so large that finding compiling programs that stop correctly or simply do not hang will be already very difficult.

The individual representation described below corresponds to the most common one¾that is, the representation that Koza used for the development of what is now standard genetic programming. In order to reduce the search space and find an individual structure adapted to the representation of a program, Koza naturally chose a functional language for several reasons:

- Syntactically speaking, with a purely functional representation, there is no need to define a grammar recognizing valid programs, provided the set of functions is closed.
- Functional languages limit side effects, which, as a side effect, minimizes bug occurrence.
- Above all, a functional program can very easily be implemented as a tree: nodes are operators that have as many children that they need operands and that return to their parent the result of their evaluation.

For instance, a function calculating a factorial:

```
Function fact(n) {
    If n=0 return 1
    Else return fact(n-1)
}
```

can be simply represented by the tree in Figure 1.

Moreover, of all different possible representations, a tree structure can naturally implement variable size individuals on which crossover operators can be very easily applied: one only needs to swap two subtrees (cf. section below on genetic operators).

Of all available functional languages, LISP is certainly the most usable. It was quite popular at the end of the 1980s, and some computers were developed around its specificities. LISP was there-