Genetic Programming

Jeroen Eggermont[a]

[a] *Division of Image Processing, Department of Radiology C2S,*
Leiden University Medical Center,
P.O. Box 9600, 2300 RC Leiden, The Netherlands

1 Evolutionary Computation

Evolutionary computation is an area of computer science which is inspired by the principles of natural evolution as introduced by Charles Darwin in "*On the Origin of Species: By Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*" [2] in 1859. As a result evolutionary computation draws much of its terminology from biology and genetics.

In evolutionary computation the principles of evolution are used to search for (approximate) solutions to problems using the computer. The problems to which evolutionary computation can be applied have to meet certain requirements. The main requirement is that the quality of that possible solution can be computed. Based on these computed qualities it should be possible to sort any two or more possible solutions in order of solution quality. Depending on the problem, there also has to be a test to determine if a solution solves the problem.

Algorithm 1 The basic form of an evolutionary algorithm.

```
initialize $P_0$
evaluate $P_0$
$t = 0$
while not stop criterion do
    $parents \leftarrow select\_parents(P_t)$
    $offspring \leftarrow variation(parents)$
    evaluate offspring (and if necessary $P_t$)
    select the new population $P_{t+1}$ from $P_t$ and offspring
    $t = t + 1$
od
```

In Algorithm 1 we present the basic form of an evolutionary algorithm. At the start of the algorithm a set or *population* of possible solutions to a problem is generated. Each of those possible solutions, also called *individuals*, is evaluated to determine how well it solves the problem. This evaluation is called the *fitness* of the individual. After the initial population has been created, the actual evolutionary process starts. This is essentially an iteration of steps applied to the population of candidate solutions.

The first step is to select which candidate solutions are best suited to serve as the *parents* for the future generation. This selection is usually done in such a way that candidate solutions with the best performance are chosen the most often to serve as parent. In the case of evolutionary computation the offspring are the result of the *variation* operator applied to the parents. Just as in biology offspring are similar but generally not identical to their parent(s). Next, these newly created individuals are evaluated to determine their fitness, and possibly the individuals in the current population are re-evaluated as well (e.g., in case the fitness function has changed). Finally, another selection takes place which determines which of the offspring (and potentially the current individuals) will form the new population. These steps are repeated until some kind of stop criterion is satisfied, usually when a maximum number of generations is reached or when the best individual is "good" enough.

2 Genetic Programming

There is no single representation for an individual used in evolutionary computation. Usually the representation of an individual is selected by the user based on the type of problem to be solved and personal preference. Historically we can distinguish the following subclasses of evolutionary computation which all have their own name:

- Evolutionary Programming (EP), introduced by Fogel et al. [3]. EP originally was based on Finite State Machines.
- Evolution Strategies (ES), introduced by Rechenberg [8] and Schwefel [9]. ES uses real valued vectors mainly for parameter optimization.
- Genetic Algorithms (GA), introduced by Holland [4]. GA uses fixed length bitstrings to encode solutions.

In 1992 Koza proposed a fourth class of evolutionary computation, named Genetic Programming (GP), in the publication of his monograph entitled "*Genetic Programming: On the Programming of Computers by Natural Selection*" [6]. In his book Koza shows how to evolve computer programs, in LISP, to solve a range of problems, among which symbolic regression. The programs evolved by Koza are in the form of parse trees, similar to those used by compilers as an intermediate format between the programming language used by the programmer (e.g., C or Java) and machine specific code. Using parse trees has advantages since it prevents syntax errors, which could lead to invalid individuals, and the hierarchy in a parse tree resolves any issues regarding function precedence.

Although genetic programming was initially based on the evolution of parse trees the current scope of Genetic Programming is much broader. In [1] Banzhaf et al. describe several GP systems using either trees, graphs or linear data structures for program evolution and in [7] Langdon discusses the evolution of data structures.

3 Tree-based Genetic Programming

The main differences between the different subclasses of evolutionary computation are their representations. In order to use a specific representation for an evolutionary algorithm one needs to specify the initialization method and variation operators. Here we discuss the (standard) initialization and variation routines we used for our tree-based Genetic Programming algorithms. We start with the initialization methods in Section 3.1. In Section 3.2 we continue with the variation process and the genetic operators.

3.1 Initialization

The first step of an evolutionary algorithm is the initialization of the population. In the case of tree-based Genetic Programming this means we have to construct syntactically valid trees. The main parameter for the initialization method is the *maximum tree depth*. This parameter is used to restrict the size of the initialized trees. Apart from the *maximum tree depth* parameter the initialization function needs a set of possible terminals T and a set of possible internal nodes I (non-terminals). For tree-based GP there are two common methods to construct trees: the *grow* method and the *full* method.

The *grow* method is given in Algorithm 2 as a recursive function returning a *node* and taking the depth of the *node* to be created as argument. The function is initially called with *depth* 0. If *depth* is smaller than the *maximum tree depth* a node is chosen randomly from the set of terminals and non-terminals. Next depending on whether *node* is a terminal (without child nodes) or internal node the *grow* method is called to create the children of *node*. If *depth* does equal *maximum tree depth* a node is chosen from the set of terminals. An example of the *grow* method with a *maximum tree depth* of 2 can be seen in Figure 1. The root node with non-terminal I_5 is created first. The

Algorithm 2 *node* **grow**(*depth*)

```
if $depth < maximum\ tree\ depth$
    $node \leftarrow random(T \cup I)$
    for $i = 1$ to $number\ of\ children$ of $node$ do
        $child_i = grow(depth+1)$
    od
else
    $node \leftarrow \mathrm{random}(T)$
fi
return $node$
```

first child of the root node becomes a terminal T_3 while the second child becomes another internal node containing non-terminal I_1. Because of the limitations set by the *maximum tree depth* both children of node I_1 become terminals.

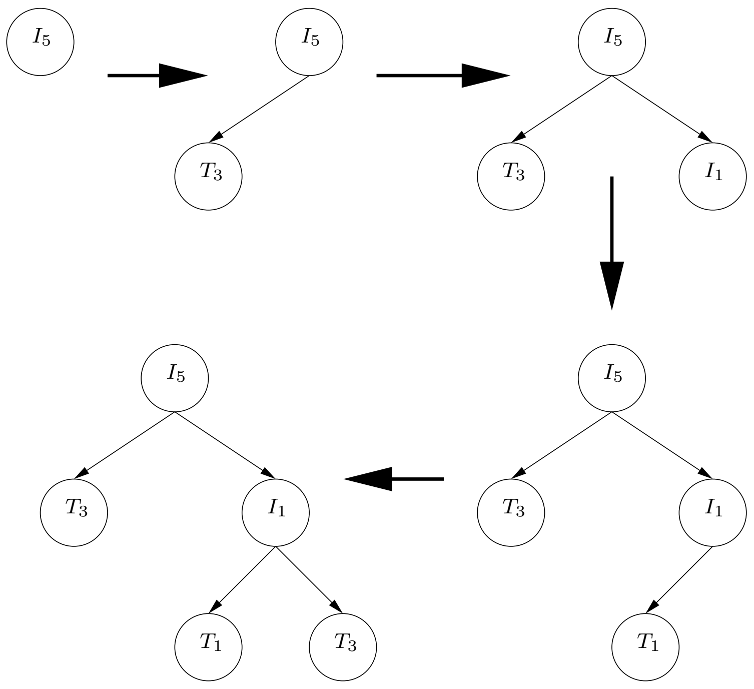

Figure 1: An example of a tree constructed using the *grow* method.

The *full* method, given in Algorithm 3, is similar to the *grow* method with one important exception. If *depth* is smaller than the maximum allowed tree depth a node is chosen randomly from the set of internal nodes I and not from the combined set of terminals (T) and non-terminals (I). An example of a tree created using the *full* method with *maximum tree depth* 2 can be seen in Figure 2. First a root node containing non-terminal I_5 is created, which has two children. The first child is created with non-terminal I_2 which has only a single child which becomes a terminal. The second child is created with non-terminal I_1 which has two terminals as children.

3.1.1 Ramped Half-and-Half Method

To ensure that there is enough diversity in the population a technique has been devised called *Ramped Half-and-Half* [5] which combines the *grow* and *full* initialization methods. Given a *maximum tree depth* D the *Ramped Half-and-Half* method divides the population size into $D-1$ groups. Each group uses a different *maximum tree depth* $(2, \ldots, D)$, whereby half of each group is created using the *grow* method and the other half using the *full* method. The result is a mix of irregular trees of different depths created by the *grow* method and more regular trees created by the *full* method.

Algorithm 3 *node* **full**(*depth*)

```
if $depth < maximum\ tree\ depth$
    $node \leftarrow$ random($I$)
    for $i = 1$ to $number\ of\ children$ of $node$ do
        $child_i = full(depth+1)$
    od
else
    $node \leftarrow$ random($T$)
fi
return $node$
```

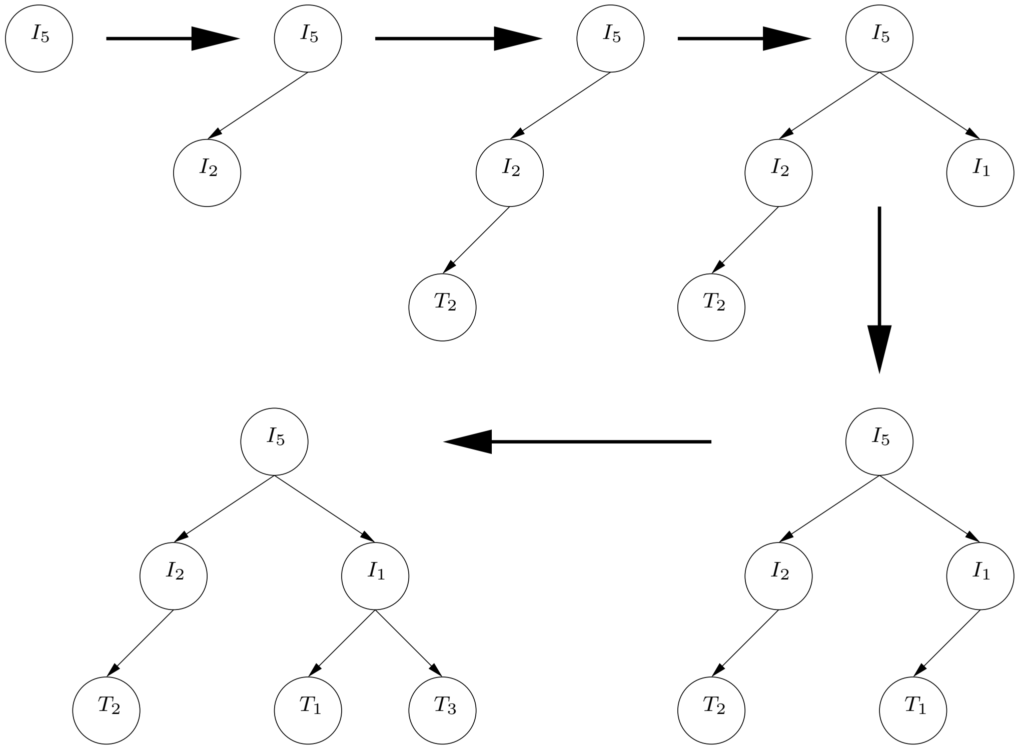

Figure 2: An example of a tree constructed using the *full* method.

3.2 Genetic Operators

After the initial population has been initialized and evaluated by a fitness function the actual evolutionary process starts. The first step of each generation is the selection of parents from the current population. These parents are employed to produce offspring using one or more genetic operators. In evolutionary computation we can distinguish between two different types of operators: crossover and mutation:

- The crossover or recombination operator works by exchanging “genetic material” between two or more parent individuals and may result in several offspring individuals.
- The mutation operator is applied to a single individual at a time and makes (small) changes in the genetic code of an individual. Mutation is often applied to the individuals produced by the crossover operator.

The combination of crossover and mutation mimics procreation in nature where the DNA of a child is a combination of the DNA of its parents whereby as a result of DNA copying errors also small mutations arise. An overview of the variation process can be seen in Figure 3.

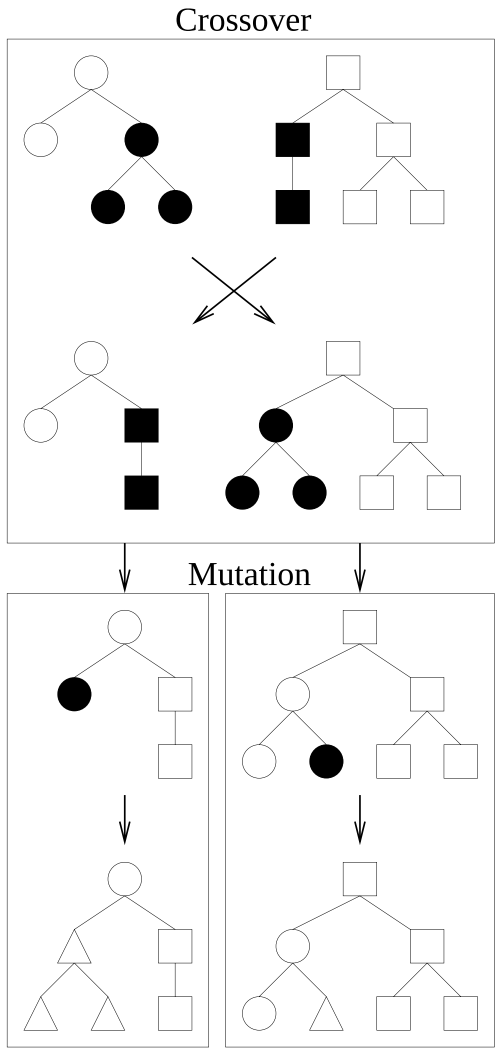

Figure 3: Overview of the variation process.

3.2.1 Crossover

There are several different crossover operators possible for tree-based Genetic Programming [1]. In this thesis we use the standard tree-crossover operator used for GP: *subtree exchange crossover*. A schematic example of *subtree exchange crossover* can be seen in Figure 4. After two parent individuals have been selected, in this case the "round" tree and the "square" tree, a subtree is randomly chosen in each of the parents. Next the two subtrees are exchanged resulting in two trees each of which is a different combination of the two parents.

3.2.2 Mutation

After the crossover operator we apply the mutation operator. In this thesis we use *subtree replacement* as mutation operator. A schematic example of *subtree replacement mutation* can be seen in Figure 5. *Subtree replacement mutation* selects a subtree in the individual to which it is applied and replaces the subtree with a randomly created tree. This subtree is usually created using one of the initialization methods described in Section 3.1.

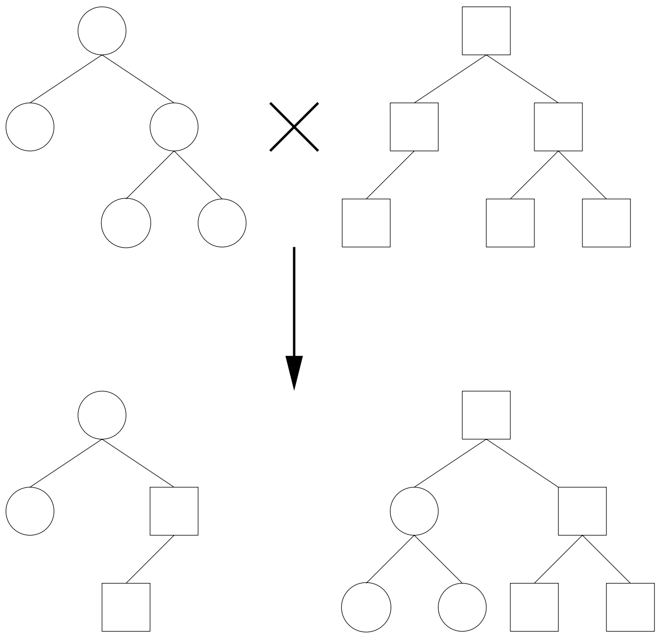
Figure 4: Subtree crossover.

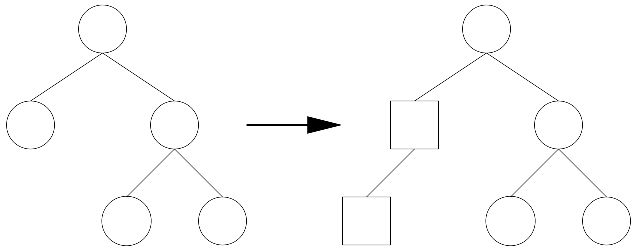
Figure 5: Subtree mutation.

References

[1] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming — An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, 1998.

[2] C. Darwin. *On the origin of species: By Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. Murray, London, 1859.

[3] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, Inc., New York, 1966.

[4] J.H. Holland. *Adaptation in natural artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[5] J.R. Koza. A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In *Proceedings of IJCNN International Joint Conference on Neural Networks*, volume IV, pages 310–318. IEEE Press, 1992.

[6] J.R. Koza. *Genetic Programming*. MIT Press, 1992.

[7] W.B. Langdon. *Genetic Programming + Data Structures = Automatic Programming!* Kluwer, 1998.

[8] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.

[9] H.-P. Schwefel. *Evolution and Optimum Seeking.* Wiley, New York, 1995.