Chapter 2

BASIC CONCEPTS OF LINEAR GENETIC PROGRAMMING

In this chapter linear genetic programming (LGP) will be explored in further detail. The basis of the specific linear GP variant we want to investigate in this book will be described, in particular the programming language used for evolution, the representation of individuals, and the specific evolutionary algorithm employed. This will form the core of our LGP system, while fundamental concepts of linear GP will also be discussed, including various forms of program execution.

Linear GP operates with imperative programs. All discussions and experiments in this book are conducted independently from a special type of programming language or processor architecture. Even though genetic programs are interpreted and partly noted in the high-level language C, the applied programming concepts exist principally in or may be translated into most modern imperative programming languages, down to the level of machine languages.

2.1 Representation of Programs

The imperative programming concept is closely related to the underlying machine language, in contrast to the functional programming paradigm. All modern CPUs are based on the principle of the von Neumann architecture, a computing machine composed of a set of registers and basic instructions that operate and manipulate their content. A program of such a register machine, accordingly, denotes a sequence of instructions whose order has to be respected during execution.

```
void gp(r)
  double r[8];
{  ...
   r[0] = r[5] + 71;
// r[7] = r[0] - 59;
   if (r[1] > 0)
   if (r[5] > 2)
     r[4] = r[2] * r[1];
// r[2] = r[5] + r[4];
   r[6] = r[4] * 13;
   r[1] = r[3] / 2;
// if (r[0] > r[1])
//   r[3] = r[5] * r[5];
   r[7] = r[6] - 2;
// r[5] = r[7] + 15;
   if (r[1] <= r[6])
     r[0] = sin(r[7]);
}
```

Example 2.1. LGP program in C notation. Commented instructions (marked with `//`) have no effect on program output stored in register `r[0]` (see Section 3.2.1).

Basically, an *imperative instruction* includes an operation on *operand* (or *source*) *registers* and an assignment of the result of that operation to a *destination register*. Instruction formats exist for zero,[1] one, two or three registers. Most modern machine languages are based on 2-register or 3-register instructions. Three-register instructions operate on two arbitrary registers (or constants) and assign the result to a third register, e.g., $r_i := r_j + r_k$. In 2-register instructions, instead, either the implemented operator requires only one operand, e.g., $r_i := sin(r_j)$, or the destination register acts as a second operand, e.g., $r_i := r_i + r_j$. Due to a higher degree of freedom, a program with 3-register instructions may be more compact in size than a program consisting of 2-register instructions. Here we will study 3-register instructions with a free choice of operands.

In general, at most one operation per instruction is permitted which usually has one or two operands. Note that a higher number of operators or operands in instructions would not necessarily increase expressiveness or variability of programs. Such instructions would assign the result of a more complex expression to a register and would make genetic operations more complicated.

In the LGP system described here and outlined in [21] a genetic program is interpreted as a variable-length sequence of simple C instructions. In order to apply a program solution directly to a problem domain without

[1] 0-register instructions operate on a stack.

using a special interpreter, the internal representation is translated into C code.[2] An excerpt of a linear genetic program, as exported by the system, is given in Example 2.1. In the following, the term *genetic program* always refers to the internal LGP representation that we will discuss in more detail now.

2.1.1 Coding of Instructions

In our implementation all registers hold floating-point values. Internally, constants are stored in registers that are write-protected, i.e., may not become destination registers. As a consequence, the set of possible constants remains fixed. Constants are addressed by indices in the internal program representation just like variable registers and operators. Constant registers are only initialized once at the beginning of a run with values from a user-defined range. This has an advantage over encoding constants explicitly in program instructions because memory space is saved, especially insofar as real-valued constants or larger integer constants are concerned. A continuous variability of constants by the genetic operators is really not needed and should be sufficiently counterbalanced by interpolation in the genetic programs. Furthermore, a free manipulation of real-valued constants in programs could result in solutions that may not be exported accurately. Because floating-point values can be printed only to a certain precision, rounding errors might be reinforced during program execution.

Each of the maximum of four instruction components, the instruction identifier and a maximum of three register indices, can be encoded into one byte of memory if we accept that the maximum number of variable registers *and* constant registers is restricted to 256. For most problems LGP is run on this will be absolutely sufficient.

So for instance, an instruction $r_i := r_j + r_k$ reduces to a vector of indices $< id(+), i, j, k >$. Actually, an instruction is held as a single 32-bit integer value. Such a coding of instructions is similar to a representation as machine code [90, 9] but can be chosen independently from the type of processor to interpret the program. In particular, this coding allows an instruction component to be accessed efficiently by casting the integer value which corresponds to the instruction into an array of 4 bytes. A program is then represented by an array of integers. A compact representation

[2] For the program instructions applied throughout the book translation is straightforward. Representation and translation of more advanced programming concepts will be discussed briefly later in this chapter.

like this is not only memory-efficient but allows efficient manipulation of programs as well as efficient interpretation (see Section 2.2).

In the following we will refer to a *register* only as a variable register. A constant register is identified with its constant value.

In linear GP a user-defined number of variable registers, the *register set*, is made available to a genetic program. Besides the minimal number of *input registers* required to hold the program inputs before execution, additional registers can be provided in order to facilitate calculations. Normally these so-called *calculation registers* are initialized with a constant value (e.g., 1) each time a program is executed on a fitness case. Only for special applications like time series predictions with a defined order on the fitness cases it may be advantageous to change this. Should calculation registers be initialized only once before fitness evaluation, an exchange of information is enabled between successive executions of the same program for different fitness cases.

A sufficient number of registers is important for the performance of linear GP, especially if input dimension and number of input registers are low. In general, the number of registers determines the number of program paths (in the functional representation) that can be calculated in parallel. If an insufficient number is supplied there will be too many conflicts between registers and valuable information will be overwritten.

One or more input/calculation registers are defined as *output register(s)*. The standard output register is register r_0. The imperative program structure also facilitates the use of multiple program outputs, whereas tree GP can calculate only one output (see also Section 8.1).

2.1.2 Instruction Set

The *instruction set* defines the particular programming language that is evolved. The LGP system is based on two fundamental *instruction type* – operations[3] and conditional branches. Table 2.1 lists the general notation of all instructions used in experiments throughout the book.

Two-operand instructions may either possess two indexed variables (registers) r_i as operands or one indexed variable and a constant. One-operand instructions only use register operands. This way, assignments of constant values, e.g., $r_0 := 1+2$ or $r_0 := sin(1)$, are avoided automatically (see also Section 7.3). If there cannot be more than one constant per instruction, the percentage of instructions holding a constant is equal to the propor-

[3]Functions will be identified with operators in the following.

Table 2.1. LGP instruction types.

Instruction type	General notation	Input range
Arithmetic operations	$r_i := r_j + r_k$	$r_i, r_j, r_k \in \mathbb{R}$
	$r_i := r_j - r_k$	
	$r_i := r_j \times r_k$	
	$r_i := r_j \ / \ r_k$	
Exponential functions	$r_i := {r_j}^{(r_k)}$	$r_i, r_j, r_k \in \mathbb{R}$
	$r_i := e^{r_j}$	
	$r_i := ln(r_j)$	
	$r_i := {r_j}^2$	
	$r_i := \sqrt{r_j}$	
Trigonomic functions	$r_i := sin(r_j)$	$r_i, r_j, r_k \in \mathbb{R}$
	$r_i := cos(r_j)$	
Boolean operations	$r_i := r_j \wedge r_k$	$r_i, r_j, r_k \in \mathbb{B}$
	$r_i := r_j \vee r_k$	
	$r_i := \neg\ r_j$	
Conditional branches	$if\ (r_j > r_k)$	$r_j, r_k \in \mathbb{R}$
	$if\ (r_j \leq r_k)$	
	$if\ (r_j)$	$r_j \in \mathbb{B}$

tion of constants p_{const} in programs. This is also the selection probability of a constant operand during initialization of programs and during mutations. The influence of this parameter will be analyzed in Section 7.3. In most other experiments documented in this book $p_{const} = 0.5$ will be used.

In genetic programming it must be guaranteed somehow that only valid programs are created. The genetic operators – recombination and mutation – have to maintain the *syntactic correctness* of newly created programs. In linear GP, for instance, crossover points may not be selected inside an instruction and mutations may not exchange an instruction operator for a register. To assure *semantic correctness*, partially defined operators and functions may be protected by returning a high value for undefined input, e.g., $c_{undef} := 10^6$. Table 2.2 shows all instructions from Table 2.1 that have to be protected from certain input ranges and provides their respective definition. The return of high values will act as a penalty for programs that use these otherwise undefined operations. If low values would be returned, i.e., $c_{undef} := 1$, protected instructions may be exploited more easily by evolution for the creation of semantic introns (see Section 3.2.2).

In order to minimize the input range assigned to a semantically senseless function value, undefined negative inputs have been mapped to defined

Table 2.2. Definitions of protected instructions.

Instruction	Protected definition			
$r_i := r_j \ / \ r_k$	$if\ (r_k \neq 0)$	$r_i := r_j \ / \ r_k$	$else$	$r_i := r_j + c_{undef}$
$r_i := {r_j}^{r_k}$	$if\ (\|r_k\| \leq 10)$	$r_i := \|r_j\|^{r_k}$	$else$	$r_i := r_j + r_k + c_{undef}$
$r_i := e^{r_j}$	$if\ (\|r_j\| \leq 32)$	$r_i := e^{r_j}$	$else$	$r_i := r_j + c_{undef}$
$r_i := ln(r_j)$	$if\ (r_j \neq 0)$	$r_i := ln(\|r_j\|)$	$else$	$r_i := r_j + c_{undef}$
$r_i := \sqrt{r_j}$	$r_i := \sqrt{\|r_j\|}$			

absolute inputs in Table 2.2. This permits evolution to integrate protected instructions into robust program semantics more easily. Keijzer [58] recommends the use of interval arithmetic and linear scaling instead of protecting mathematical operators for symbolic regression.

The ability of genetic programming to find a solution strongly depends on the expressiveness of the instruction set. A *complete* instruction set contains all elements that are necessary to build the optimal solution, provided that the number of variables registers and the range of constants are sufficient. On the other hand, the dimension of the search space, which contains all possible programs that can be built from these instructions, increases exponentially with the number of instructions and registers. If we take into account that the initial population usually represents a small fraction of the complete search space, the probability of finding the optimal solution or a good approximation decreases significantly with too many basic program elements that are useless. Moreover, the probability by which a certain instruction is selected as well as its frequency in the population influence the success rate of finding a solution. In order to exert better control over the selection probabilities of instruction types, the instruction set may contain *multiple instances* of an instruction.

We will not regard program functions with *side effects* to the problem environment, only those that return a single value in a strict mathematical sense. Side effects may be used for solving control problems. For instance, a linear program may represent a list of commands (plan) that direct a robot agent in an environment. Fitness information may then be derived from the agent's interactions with its environment by *reinforcement learning*. In such a case, genetic programs do not represent mathematical functions.

2.1.3 Branching Concepts

Conditional branches are an important and powerful concept in genetic programming. In general, programming concepts like branches or loops allow the control flow given by the structure of the representation to

be altered. The control flow in linear genetic programs is linear while the data flow is organized as a directed graph (see Section 3.3). With conditional branches the control flow (and hence the data flow) may be different for different input situations, for instance, it may depend on program semantics.

Classification problems are solved more successfully or even exclusively if branches are provided. Branches, however, may increase the complexity of solutions by promoting specialization and by producing semantic introns (see Chapter 3). Both tendencies may lead to less robust and less general solutions.

If the condition of a branch instruction, as defined in Table 2.1, is false only *one* instruction is skipped (see also discussion in Section 3.3.2). Sequences of branches are interpreted as *nested branches* in our system (similar to interpretation in C). That is, the next non-branch instruction in the program is executed only if all conditions are true and is skipped otherwise. A combination of conditional branch(es) and operation is also referred to as a *conditional operation*:

```
if (<cond1>)
if (<cond2>)
<oper>;
```

Nested branches allow more complex conditions to be evolved and are equivalent to connecting single branch conditions by a logical AND. A disjunction (OR connection) of branch conditions, instead, may be represented by a sequence of conditional instructions whose operations are identical:

```
if (<cond1>)
<oper>;
if (<cond2>)
<oper>;
```

Alternatively, successive conditions may be interpreted as being connected either by AND or by OR. This can be achieved in the following way: A Boolean operator is encoded into each branch identifier. This requires the information of a binary flag only, which determines how the condition of a branch instruction is connected to a potentially preceeding or, alternatively, succeeding one in the program (AND or OR). The status of these flags may be changed during operator mutations. The transformation of

this representation into a C program becomes slightly more complicated because each sequence of branches has to be substituted by a single branch with an equivalent condition of higher order.

2.1.4 Advanced Branching Concepts

A more general branching concept is to allow conditional forward jumps over a variable number of instructions. The number of instructions skipped may be either unlimited or it may be selected randomly from a certain range. In the latter case the actual length of a jump may be determined by a parameter that is encoded in the branch instruction itself, e.g., using the identifier section or the unused section of the destination register. It is also possible to do without this additional overhead by using constant block sizes. Because some instructions of a skipped code block are usually not effective, evolution may control the semantic effect of a jump over the number of noneffective instructions within jump blocks.

A transformation of such branches from the internal program representation into working C code requires constructions like:

```
if (<cond>) goto <label X>;
<...>
<label X>;
```

where *unique* X labels have to be inserted at the end of each jump block.

If one wants to avoid branching into blocks of other branches, jumps should not be longer than the position of the *next* branch in a program. In this way, the number of skipped instructions is limited implicitly and does not have to be administrated within the branches. Translation into C is then achieved simply by setting `{...}` brackets around the jump block.

An interesting variant of the above scheme is to allow jumps to *any* succeeding branch instruction in the program. This can be realized by using an additional pointer with each branch instruction to an arbitrary successor branch (*absolute jump*). *Relative jumps* to the kth *next* branch in program with $1 \leq k \leq k_{max}$ are also possible, even if such connections are separated more easily by insertions/deletions of a new branch instruction. A pointer to a branch that does not exist any more may be automatically replaced by a valid pointer after variation. The last branch in a program should always point to the end of the program ($k := 0$). Hence, control flow in a linear genetic program may be interpreted as a *directed acyclic*

branching graph (see Figure 2.1). The nodes of such a *control flow graph* represent subsequences of (non-branch) instructions.

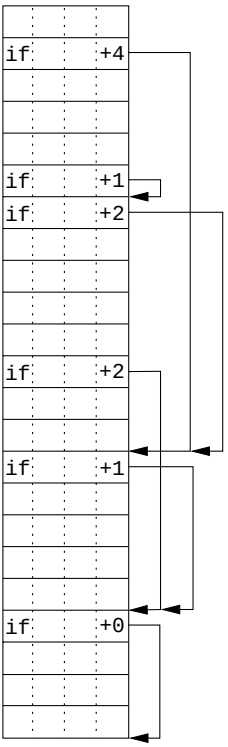

Figure 2.1. Branching graph: Each branch instruction points to a specified succeeding branch instruction.

In [57] a more general concept of a branching graph is proposed for the imperative representation. Each node contains an instruction block that ends with a single *if-else*-branch. These branches point to two alternative decision blocks which represent two independent successor nodes. Thus, instructions may not only be skipped within an otherwise linear control flow but real parallel subprograms may exist in programs. This form of representation is called a *linear graph* since it defines a graph-based control flow on linear genetic programs. Recall that the term *linear genetic program* derives from the linear flow of control that is given by the linear arrangement of instructions. In Section 3.3 we will see that the data flow is graph-based already in simple linear genetic programs.

In general, a complex non-linear control flow requires either more sophisticated variation operators or repair mechanisms to be applied after variation. For branching graphs a special crossover operator may be con-

strained so that only complete nodes or subgraphs of nodes are exchanged between programs with a certain probability. That is, crossover points would fall upon branch instructions only. Unrestricted linear crossover (see Section 2.3.4) may be applied between graph nodes (instruction blocks) only.

A final branching concept whose capability is discussed here for linear GP uses an additional `endif` instruction in the instruction set. Nested constructions like:

```
if (<cond>)
<...>
endif
```

are interpreted such that an `endif` belongs to an `if` counterpart if *no* branch or *only closed* branching blocks lie in between. An instruction that cannot be assigned in this way may either be deleted from the internal representation or contribute to noneffective code. The strength of such a concept is to permit an (almost) unconstrained and complex nesting of branches while jumps into other branching blocks cannot occur. A transformation into C code is achieved simply by setting `{...}` brackets around valid branching blocks instead of `endif` and by not transforming invalid branch instructions at all. In a similar way `if-else-endif` constructions may be realized.

2.1.5 Iteration Concepts

Iteration of code by loops plays a rather unimportant role in genetic programming. Most GP applications that require loops involve control problems with the combination of primitive actions of an agent being the object of evolution. Data flow is usually not necessary in such programs. Instead, each instruction performs actions with side effects on the problem environment and fitness is derived from a reinforcement signal. For the problem classes we focus on here, supervised classification and approximation, iteration is of minor importance. That is not to say that a reuse of code by iterations could not result in more compact and elegant solutions.

In functional programming the concept of loops is unknown. The implicit iteration concept in functional programs denotes *recursions* which are, however, hard to control in tree-based genetic programming [142]. Otherwise, iterated evaluations of a subtree can have an effect only if functions produce side effects. In linear GP, assignments represent an implicit side

effect on memory locations as part of the imperative representation. Nevertheless, the iteration of an instruction segment may only be effective if it includes at least one effective instruction *and* if at least one register acts as both destination register and source register in the same or a combination of (effective) instructions, e.g., $r_0 := r_0 + 1$.

In the following, possible iteration concepts for linear GP will be presented. These comprise *conditional loops* and loops with a *limited* number of iterations.

One form of iteration in linear programs is a conditional backward jump corresponding to a `while` loop in C. The problem with this concept is that infinite loops can be easily formed by conditions that are always fulfilled. In general, it is not possible to detect all infinite loops in programs, due to the *halting problem* [36]. A solution to remedy this situation is to terminate a genetic program after a maximal number of instructions. The result of the program would then, however, depend on the execution time allowed.

The more recommended option is a loop concept that limits the number of iterations in each loop. This requires an additional control flow parameter which may either be constant or be varied within loop instructions. Such a construct is usually expressed by a `for` loop in C. Because only overlapping loops (not nested loops) need to be avoided, an appropriate choice to limit the size of loop blocks may be the coevolution of `endfor` instructions. Analogous to the interpretation of branches in Section 2.1.4, a `for` instruction and a succeeding `endfor` define a loop block provided that *only closed loops* lie in between. All other loop instructions are not interpreted.

2.1.6 Modularization Concepts

For certain problems modularization may be advantageous in GP. By using subroutines repeatedly within programs, solutions may become more compact and the same limited program space can be used more efficiently. A problem may also be decomposed into simpler subproblems that can be solved more efficiently in local submodules. In this case, a combination of subsolutions may result in a simpler and better overall solution.

The most popular modularization concept in tree-based genetic programming is the so-called *automatically defined function* (ADF) [65]. Basically, a genetic program is split up into a main program and a certain number of subprograms (ADFs). The main program calculates its output by using the coevolved subprograms via function calls. Therefore, ADFs are treated as part of the main instruction set. Each module type may be com-

posed of different sets of program components. It is furthermore possible to define a usage graph that defines which ADF type may call which other ADF type. Recursions are avoided by prohibiting cycles. The crossover operator has to be constrained in such a way that only modules of the same type can be recombined between individuals.

ADFs are an *explicit* modularization concept since the submodules are encapsulated with regard to the main program and may only be used locally in the same individual. Each module is represented by a separate tree expression [65] or a separate sequence of instructions [93]. To ensure encapsulation of modules in linear programs, disjoint sets of registers have to be used. Otherwise, unwanted state transitions between modules might occur.

ADFs denote subsolutions that are combined by being used in a main program. In Chapter 11 of this book another explicit form of modularization, the evolution of program *teams*, is investigated. A team comprises a fixed number of programs that are coevolved as one GP individual. In principle, all members of a team are supposed to solve the same problem by receiving the same input data. These members act as modules of an overall solution such that the member outputs are combined in a predefined way. A better performance may result from collective decision making and a specialization of relatively independent program modules.

A more *implicit* modularization concept that prepares code for reuse is *module acquisition* [5]. Here substructures up to a certain maximum size – not only including full subtrees – are chosen randomly from better programs. Such modules are replaced by respective function calls and moved into a global library from where they may be referenced by other individuals of the population. In linear GP code replacements are more complicated because subsequences of instructions are usually bound to a complex register usage in the imperative program context.

A similar method for automatic modularization is *subtree encapsulation* [115] where randomly selected subtrees are replaced by symbols that are added to the terminal set as primitive atoms.

Complex module dependencies may hardly emerge during evolution if modularization is not really needed for better solutions. In general, if a programming concept is redundant, the larger search space will negatively influence the ability to find a solution. Moreover, the efficiency of a programming concept or a program representation in GP always depends on the variation operators. Thus, even if the expressiveness or flexibility of a programming concept is high, it may be more difficult for evolution to take advantage of that strength.

2.2 Execution of Programs

The processing speed of a learning method may seriously constrain the complexity or time-dependence of an application. The most time-critical steps in evolutionary algorithms are the fitness evaluation of individuals and/or the calculation of new search points (individuals) by variation operators. In genetic programming, however, computation costs are dominated by the fitness evaluation because it requires multiple executions of a program, at least one execution per fitness case. Executing a genetic program means that the internal program representation is interpreted following the semantics of the programming language.

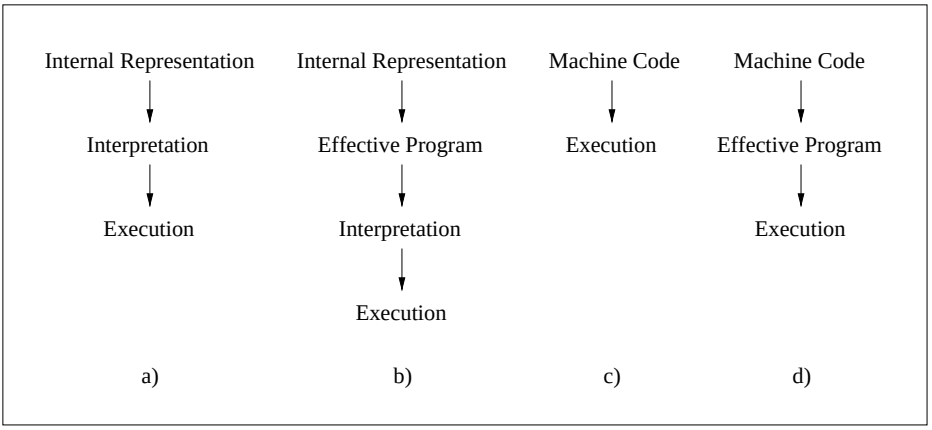

Figure 2.2. Different forms of program execution including (a) interpretation of programs in GP, (b) elimination of noneffective code in LGP, (c) direct execution of machine code in AIMGP, and (d) combination of b) and c).

For instance, interpretation in tree GP systems works by traversing the tree structure of programs in preorder or postorder. While doing so, operators are applied to operand values that result recursively from executing all subtrees of the operator node first.

In a special variant of linear GP, called *Automatic Induction of Machine code by Genetic Programming* (AIMGP) [90, 9], individuals are represented and manipulated as binary machine code. Because programs can be executed directly without passing an interpreter, machine code GP enjoys a significant speedup in execution compared to interpreting GP systems. Due to its dependence on specific processor architectures, however, machine code GP is restricted in portability. Moreover, a machine code system may be restricted in functionality due to, e.g., the number of hardware registers resident in the processor.

In this book we use a different method to accelerate execution (interpretation) of linear genetic programs. The special type of noneffective code which results from the imperative program structure can be detected efficiently in linear runtime (see [21] and Section 3.2.1). In LGP this noneffective code is removed from a program before fitness calculation, i.e., before the resulting effective program is executed over multiple fitness cases. By doing so, the evaluation time of programs is reduced significantly, especially if a larger number of fitness cases is to be processed (see below and Chapter 4). In the example program from Section 2.1 all commented instructions are noneffective under the assumption that program output is stored in register `r[0]`.

Since AIMGP is a special variant of linear GP, both acceleration techniques may be combined such that a machine code representation is preprocessed by a routine extracting effective parts of code. This results in the four different ways of executing programs in genetic programming that are illustrated in Figure 2.2.

An elimination of introns – as noneffective code is frequently called – will be relevant only if a significant amount of this code is created by the variation operators. In particular, this is the case for linear crossover (see Section 2.3.4).

An additional acceleration of runtime in linear GP can be achieved if the fitness of an individual is recalculated only after effective code has undergone change (see Section 5.2). Instead of the evaluation *time*, this approach reduces the *number* of evaluations (and program executions) performed during a generation.

2.2.1 Runtime Comparison

The following experiment illustrates the difference in processing speed of the four ways of program execution depicted in Figure 2.2. In order to guarantee a fair comparison between machine code GP and interpreting GP, an interpreting routine has been added to an AIMGP system. This routine *interprets* the machine code programs in C so that they produce exactly the same results as without interpretation. Both interpreting and non-interpreting runs of the system are accelerated by a second routine that removes the noneffective code. Table 2.3 reports general settings of system parameters for a polynomial regression task.

Table 2.3. Parameter settings

Parameter	Setting
Problem type	polynomial regression
Number of examples	200
Number of generations	200
Population size	1,000
Maximum program length	256
Maximum initial length	25
Crossover rate	90%
Mutation rate	10%
Number of registers	6
Instruction set	{+,−,×}
Constants	{0,..,99}

Table 2.4 compares the average *absolute* runtime[4] for the four different configurations with respect to interpretation and intron elimination. Without interpretation, programs are executed directly as machine code. Ten runs have been performed for each configuration while using the same set of 10 different random seeds. Runs behave exactly the same for all configurations apart from their processing speed. The average length of programs in the population exceeds 200 instructions by about generation 100. The intron rate converges to about 80% on average.

Table 2.4. Absolute runtime in seconds (rounded) averaged over 10 independent runs.

Runtime (sec.)	No Interpretation (I_0)	Interpretation (I_1)
No Intron Elimination (E_0)	500	6250
Intron Elimination (E_1)	250	1375

The resulting *relative* acceleration factors are listed in Table 2.5. If both the direct execution of machine code *and* the elimination of noneffective code are applied in combination, runs become about 25 times faster for the problem considered under the system configuration above. Note that the influence of intron elimination on the interpreting runs (factor 4.5) is more than two times stronger than on the non-interpreting runs (factor 2). This reduces the advantage of machine code GP over interpreting LGP from a factor of 12.5 to a factor of 5.5. Standard machine code GP without intron elimination, instead, seems to be around 3 times faster than linear GP including this extension.

[4] Absolute runtime is measured in seconds on a Sun SPARC Station 10.

Table 2.5. Relative runtime for the four configurations of Table 2.4.

$E_0I_0 : E_0I_1$	1 : 12.5
$E_1I_0 : E_1I_1$	1 : 5.5
$E_0I_0 : E_1I_0$	1 : 2
$E_0I_1 : E_1I_1$	1 : 4.5
$E_0I_0 : E_1I_1$	1 : 2.75
$E_1I_0 : E_0I_1$	1 : 25

Clearly, the performance gain by intron elimination will depend on the proportion of (structurally) noneffective instructions in programs. In contrast to the size of effective code, this is less influenced by the problem definition than by variation operators and system configuration (see Chapters 5 and 7).

2.2.2 Translation

From an application point of view the best (generalizing) program solution is the only relevant result of a GP run. The internal representation (coding) of this program could be exported as is and an interpreter would be required to guarantee that the program will behave in an application environment exactly as it did in the GP system. In order to avoid this, LGP exports programs as equivalent C functions (see Example 2.1 and

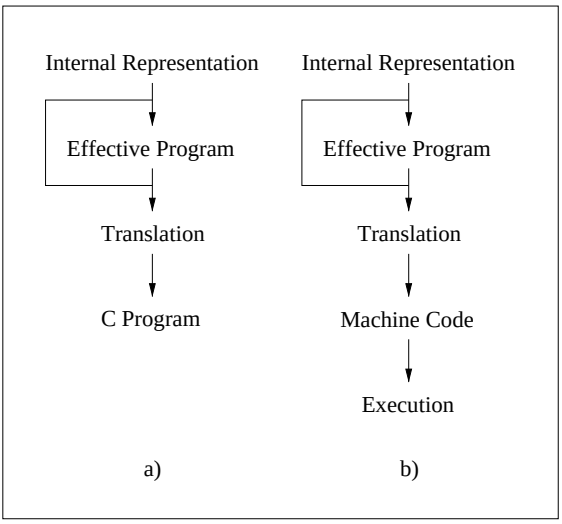

Figure 2.3. Translation of internal representation into (a) C program and (b) machine code.

Figure 2.3). As has been explained in Section 2.1.2, single programming concepts are transformed into C by translating internal programs into an existing (imperative) programming language. This way, solutions may be integrated directly into an application context (software) without additional overhead.

Such a translation has the additional benefit to allow more freedom on the internal representation. The representation may be chosen (almost) freely, e.g., in favor of better evolvability and better variability in GP. Because normally just a few (best) individuals are exported during a run, even complex transformations may not be time-critical.

The same advantage – higher flexibility – together with a higher processing speed motivates a translation from the evolved LGP language into a binary machine language (*compilation*) just before the fitness of a program is evaluated (see Figure 2.3). This allows a more efficient evaluation of programs, especially if noneffective code is removed prior to translation. Note that the direct variation of machine code programs in AIMGP systems is less important for runtime. Instead, the speed advantage almost exclusively results from a direct execution of machine code. The disadvantage of this technique is the higher compiler overhead that needs to be taken into account.

2.3 Evolution of Programs

Algorithm 2.1 constitutes the kernel of our LGP system. In a *steady-state* evolutionary algorithm like this, generations are not fixed, in contrast to a *generational* EA. For the latter variant, the current generation is identified with a population of parent programs whose offspring migrate to a *separate* population pool. After the offspring pool is fully populated it replaces the parent population and the next generation begins. In the steady-state model there is no such centralized control of generations. Instead, offspring replace existing individuals in the *same* population. It is common practice to define *generation equivalents* in steady-state EAs as regular intervals of fitness evaluations. Only new individuals have to be evaluated if the fitness is saved with each individual in the population. A generation (equivalent) is complete if the number of new individuals equals the population size.

Algorithm 2.1 (*LGP algorithm*)

1. *Initialize* the population with random programs (see Section 2.3.1) and calculate their fitness values.

2. Randomly *select* $2 \times n_{ts}$ individuals from the population without replacement.

3. Perform two *fitness tournaments* of size n_{ts} (see Section 2.3.2).

4. Make *temporary copies* of the two tournament winners.

5. *Modify* the two winners by one or more variation operators with certain probabilities (see Section 2.3.4).

6. *Evaluate* the fitness of the two offspring.

7. If the current best-fit individual is replaced by one of the offspring *validate* the new best program using unknown data.

8. *Reproduce* the two tournament winners within the population with a certain probability or under a certain condition by replacing the two tournament losers with the temporary copies of the winners (see Section 2.3.3).

9. Repeat steps 2 to 8 until the maximum number of generations is reached.

10. *Test* the program with minimum validation error.

11. Both the best program during training and the best program during validation define the output of the algorithm.

Fitness of an individual program is computed by an *error function* on a set of input-output examples $(\vec{i_k}, o_k)$. These so-called *fitness cases* define the problem that should be solved or approximated by a program. A popular error function for approximation problems is the *sum of squared errors* (SSE), i.e., the squared difference between the predicted output $gp(\vec{i_k})$ and the desired output o_k summed over all n training examples. A squared error function penalizes larger errors more heavily than smaller errors. Equation 2.1 defines a related measure, the *mean square error* (MSE).

$$\text{MSE}(gp) = \frac{1}{n} \sum_{k=1}^{n} (gp(\vec{i_k}) - o_k)^2 \tag{2.1}$$

For classification tasks the *classification error* (CE) calculates the number of examples wrongly classified. Function *class* in Equation 2.2 hides the classification method that maps the continuous program outputs to

discrete class identifiers. While a better fitness means a smaller error the best fitness is 0 in both cases.

$$\mathrm{CE}(gp) = \sum_{k=1}^{n} \{1 \mid class(gp(\vec{i_k})) \neq o_k\} \tag{2.2}$$

The generalization ability of individual solutions is observed *during* training by calculating a *validation error* of the current best program. The training error function is applied to an unknown *validation data set* which is sampled differently from the training data, but from the same data space. Finally, among all the best individuals emerging over a run the one with minimum validation error (point of best generalization) is tested on an unknown *test data set*, once *after* training. Note that validation of the best solutions follows a fitness gradient. Validating all individuals during a GP run is not reasonable, since one is not interested in solutions that perform well on the validation data but have a comparatively bad fitness on the training data set.

Whether an individual is selected for variation or ruled out depends on relative fitness comparisons during selection. In order to not loose information, a copy of the individual with minimum validation error has to be kept outside of the population. The individual of minimum training error (best individual) does not need protection since it cannot be overwritten as long as the training data is fixed during evolution.

Training data may be resampled every mth generation or even each time before an individual is evaluated. On the one hand, resampling introduces noise into the fitness function (*dynamic fitness*). This is argued to improve the generalization performance compared to keeping the training examples constant over a run because it reduces *overtraining*, i.e., an overspecialization of solutions to the training data. On the other hand, resampling may be beneficial if the database that constitutes the problem to be solved is large. A relatively small subset size may be used for training purposes while all data points would be exposed to the genetic programs over time. As a result, not only the fitness evaluation of programs is accelerated but the evolutionary process may converge faster. This technique is called *stochastic sampling* [9].

2.3.1 Initialization

The initial population of a genetic programming run is normally generated randomly. In linear GP an upper bound for the initial program length has to be defined. The lower bound may be equal to the absolute minimum length of a program – one instruction. A program is created so that

its length is chosen randomly from this predefined range with a uniform probability.

There is a trade-off to be addressed when choosing upper and lower bounds of program length: On the one hand, it is not recommended to initialize exceedingly long programs, as will be demonstrated in Section 7.6. This may reduce their variability significantly in the course of the evolutionary process. Besides, the smaller the initial programs are, the more thorough an exploration of the search space can be performed at the beginning of a run. On the other hand, the average initial length of programs should not be too small, because a sufficient diversity of the initial genetic material is necessary, especially in smaller populations or if crossover dominates variation.

2.3.2 Selection

Algorithm 2.1 applies *tournament selection*. With this selection method individuals are selected randomly from the population to participate in a tournament where they compete for the best fitness. Normally selection happens without replacement, i.e., all individuals of a tournament must be different. The *tournament size* n_{ts} determines the selection pressure that is imposed on the population individuals. If a tournament is held between *two* individuals (and if there is only one tournament used for selecting the winner) this corresponds to the minimum selection pressure. A lower pressure is possible with this selection scheme only by performing $m > 1$ tournaments and choosing the *worst* among the m winners.

In standard LGP *two* tournaments happen in parallel to provide two parent individuals for crossover. For comparison purposes, this is practiced here also in cases where only mutation is applied (see Chapter 6). Before the tournament winners undergo variation, a copy of each winner replaces the corresponding loser. This reproduction scheme constitutes a steady-state EA.

Tournament selection, together with a steady-state evolutionary algorithm, is well suited for parallelization by using isolated subpopulations of individuals, called *demes* (see also Section 4.3.2). Tournaments may be performed independently of each other and do not require global information about the population, like a global fitness ranking (*ranking selection*) or the average fitness (*fitness proportional selection*) [17] would do. Local selection schemes are arguably better to preserve diversity than global selection schemes. Moreover, individuals may take part in a tournament several times or not at all during one steady-state generation. This al-

lows evolution to progress with different speeds in different regions of the population.

2.3.3 Reproduction

A full reproduction of winners guarantees that better solutions always survive in a steady-state population. However, during every replacement of individuals a certain amount of genetic material gets lost. When using tournament selection this situation can be influenced by the *reproduction rate* p_{rr}. By using $p_{rr} < 1$ the EA may *forget* better solutions to a certain degree. Both reproduction rate and selection pressure (tournament size) have a direct influence on the convergence speed of the evolutionary algorithm as well as on the loss of (structural and semantic) diversity.

The reproduction rate could also be allowed to exceed the standard setting 1 ($p_{rr} > 1$). An individual would then be reproduced *more than once* within the population. As a result, both the convergence speed and the loss of diversity will be accelerated. Obviously, too many replications of individuals lead to an unwanted premature convergence and subsequent stagnation of the evolutionary process. Note that more reproductions are performed than new individuals are created.

Instead of, or in addition to, an explicit reproduction probability, implicit conditions can be used to determine when reproduction shall take place (see Section 10.5).

2.3.4 Variation

Genetic operators change the contents and the size of genetic programs in a population. Figure 2.4 illustrates *two-point linear crossover* as it is used in linear GP for recombining two genetic programs [9]. A segment of random position and arbitrary length is selected in each of the two parents and exchanged. In our implementation (see also Section 5.7.1) crossover exchanges equally sized segments if one of the two children would exceed the maximum length otherwise [21].

Crossover is the standard *macro operator* applied to vary the length of linear genetic programs on the level of instructions. In other words, instructions are the smallest units to be changed. *Inside* instructions *micro mutations* randomly replace either the instruction identifier, a register or a constant (if existent) by equivalents from predefined sets or valid ranges. In Chapter 5 and Chapter 6 we will introduce more advanced genetic operators for the linear program representation.

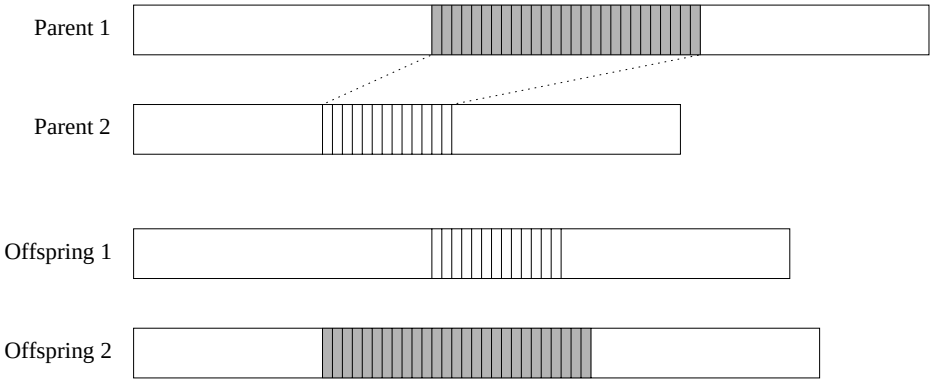

Figure 2.4. Crossover in linear GP. Continuous sequences of instructions are selected and exchanged between parents.

In general, there are three different ways in which variation operators may be selected and applied to a certain individual program before its fitness is calculated:

- ☐ Only *one* variation is performed per individual.
- ☐ *One* variation operator is applied *several* times.
- ☐ *More than one* variation operator is applied.

The advantage of using only one genetic operation per individual is a lower total variation strength. This allows artificial evolution to progress more specifically and in smaller steps. By applying several genetic operations concurrently, on the other hand, computation time is saved such that less evaluations are necessary. For example, micro mutations are often applied together with a macro operation.

Note that in all three cases, there is only one offspring created per parent individual, i.e., only one offspring gets into the population and is evaluated. However, similar to multiple reproduction of parents one may generate more than one offspring from a parent. Both options are, however, not realized by Algorithm 2.1.