Chapter 2
Grammatical Evolution

Genetic Programming (GP) is a population-based search method based upon neo-Darwinian principles of evolution, which manipulates executable structures. It can be used to automatically generate computer code to solve real-world problems in a broad array of application domains [173].

Grammatical Evolution(GE) (see [154, 148, 24, 54, 55, 151, 190, 160, 86, 25, 43, 7, 163]) is a grammar-based form of GP. It marries principles from molecular biology to the representational power of formal grammars. GE's rich modularity gives a unique flexibility, making it possible to use alternative search strategies, whether evolutionary, or some other heuristic (be it stochastic or deterministic) and to radically change its behaviour by merely changing the grammar supplied. As a grammar is used to describe the structures that are generated by GE, it is trivial to modify the output structures by simply editing the plain text grammar. The explicit grammar allows GE to easily generate solutions in any language (or a useful subset of a language). For example, GE has been used to generate solutions in multiple languages including Lisp, Scheme, C/C++, Java, Prolog, Postscript, and English. The ease with which a user can manipulate the output structures by simply writing or modifying a grammar in a text file provides an attractive flexibility and ease of application not as readily enjoyed with the standard approach to Genetic Programming. The grammar also implicitly provides a mechanism by which type information can be encoded thus overcoming the property of closure, which limits the traditional representation adopted by Genetic Programming to a single type. The genotype-phenotype mapping also means that instead of operating exclusively on solution trees, as in standard GP, GE allows search operators to be performed on the genotype (e.g., integer or binary chromosomes), in addition to partially derived phenotypes, and the fully formed phenotypic derivation trees themselves. As such, standard GP tree-based operators of subtree-crossover and subtree-mutation can be easily adopted with GE.

In this chapter an introduction to GE is given including a description of the genotype-to-phenotype mapping process in Section 2.2, which is a key

I. Dempsey et al.: Foundations in Gramm. Evol. for Dyn. Environ., SCI 194, pp. 9–24.
springerlink.com

feature of GE in dealing with dynamic environments. Section 2.3 describes how the mutation and crossover genetic operators are applied in GE, and is followed by an introduction to the adoption of alternative search engines in Section 2.4. Section 2.5 details various applications of GE. The chapter then closes with some brief conclusions.

2.1 An Overview of Grammatical Evolution

GE is a grammar-based Genetic Programming paradigm that is capable of evolving programs or rules in any language [148, 151, 154, 190]. GE is similar to Genetic Programming as popularised by Koza [110], in that it uses an evolutionary process to automatically generate variable-length computer programs. Unlike GP however, GE adopts a population of linear genotypic binary or integer strings, which are transformed into functional phenotypic programs through a genotype-to-phenotype mapping process [9]. This transformation is governed through the use of a BNF grammar, which specifies the language of the produced solutions. The mechanics of the mapping process are discussed in more detail in Section 2.2.

The mapping process creates a distinction between the search space and the solution space. Genotype strings are evolved with no knowledge (or respect) of their phenotypic equivalent, apart from a fitness measure, in an unconstrained evolutionary search. These genotypic individuals are then projected into the constrained phenotypic solution space through the use of the mapping process. In undergoing evolution and reproduction, the canonical genetic operators of Holland's GA may be applied to the genotype strings. The application of these operators has been shown to create a phenotypic "ripple effect". Evidence suggests that this effect promotes a useful exchange of derivation sub-sequences during crossover events [154], while by the same measure contributing to phenotypic diversity.

GE is modular in design, where the grammar, the search engine and the objective function all represent plug-in components to GE. Figure 2.1 presents the modular design of GE. Such modularity allows it to be used in conjunction with researchers' prefered search algorithms. In recent developments GE has been combined with a swarm algorithm [163] and a differential evolution algorithm [162]. In other developments, the potential of the grammar component has been explored with meta-Grammars [156] where a universal grammar, or grammar's grammar is defined, allowing the evolution of a solution's own vocabulary. meta-Grammars have also been used in the development of the meta-Grammar GA (mGGA) [159] that encourages the evolution of building blocks (see Section 3.3.5) which can be reused to construct solutions more efficiently. More detail on meta-Grammars combined with GE is given in Chapter 6. Also on research into grammars, attribute grammars have been examined [42, 158] that enable the addition of semantic and context-sensitive information into the grammars.

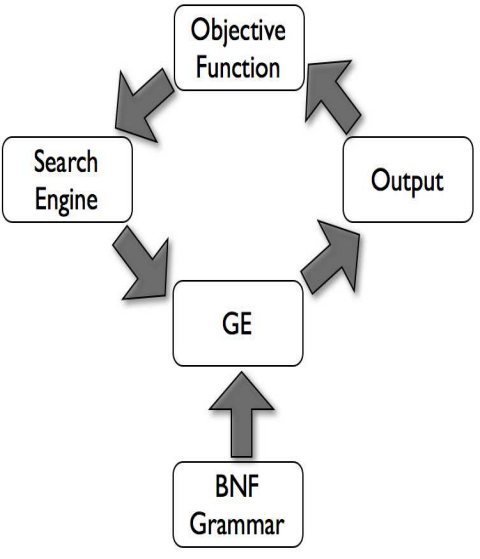

Fig. 2.1 GE's modular design and workflow

2.2 Mapping Process

When approaching a problem using GE, initially a BNF grammar must be defined. This grammar specifies the syntax of desired phenotypic programs to be produced by GE. The syntax at its broadest may detail the specification of a programming language such as C, or more practically a subset of such a language. The development of a BNF grammar also affords the researcher the ability to incorporate domain biases or domain-specific functions.

A BNF grammar is made up of the tuple N, T, P, S; where N is the set of all non-terminal symbols, T is the set of terminals, P is the set of production rules that map N to T, and S is the initial start symbol and a member of N. Where there are a number of production rules that can be applied to a non-terminal, a " | " (or) symbol separates the options. Using

```
N = {<expr>, <op>, <operand>, <var>}
T = {1, 2, 3, 4, +, -, /, *, x, y}
S = {<expr>}
```

with P, below is an example of a BNF grammar:

```
<expr>::=  <expr><op><expr>        (0)
         | <operand>               (1)
<op>::=  +                         (0)
       | -                         (1)
       | /                         (2)
       | *                         (3)
<operand>::=  1                    (0)
```

```
                | 2                    (1)
                | 3                    (2)
                | 4                    (3)
                | <var>                (4)
<var>::=  x                            (0)
        | y                            (1)
```

Using such a grammar as an input, GE then employs the expression

$$Rule = c\%r$$

that selects an option from the production rule for the symbol being currently mapped; where c is the codon value and r is the number of production rules available for the current non-terminal. An example of the mapping process employed by GE is shown in Figure 2.2. Starting with a binary string, an integer string is evolved, typically by using 8 bits per codon, which is a sequence of genes. These are then used to choose rules from a given BNF grammar and to generate a phenotypic program.

Beginning with a given *start symbol* `<E>`, a production associated with that symbol is chosen to replace it, by using the current codon from the integer string. In the example, the codon 8 is mapped to the number of available productions associated with `<E>`, of which there are 6. So therefore $8 \; mod \; 6 = 2$, and `<E>` is replaced with the sequence `(- <E><E>)`.

The next step consists of choosing a production for the next `<E>` non-terminal symbol, which is now the leftmost symbol on the phenotype string under construction. This is done by using the next codon, 4, and again mapping it to the number of productions associated with `<E>`, giving $4 \; mod \; 6 = 4$, so `<E>` is replaced with `(+ <E><E>)`.

The mapping process continues in this manner, always replacing the leftmost non-terminal symbol with a production associated with that symbol on the grammar, chosen by a codon. The mapping in standard GE terminates when one of the following conditions is met:

1. A complete program is generated before the end of the genome is encountered. This occurs when all the non-terminals in the expression being mapped are transformed into elements from the terminal set of the BNF grammar as seen in the example in Figure 2.2.
2. The end of the genome is reached, in which case the *wrapping* operator [155] is invoked. This results in the return of the genome-reading frame to the left-hand side of the genome once again. The reading of codons will then continue as before unless an upper threshold, representing the maximum number of wrapping events, is reached during the mapping process.
3. A threshold on the number of wrapping events is reached and the individual is still not completely mapped. In this case the mapping process is halted and the individual is assigned the worst possible fitness value.

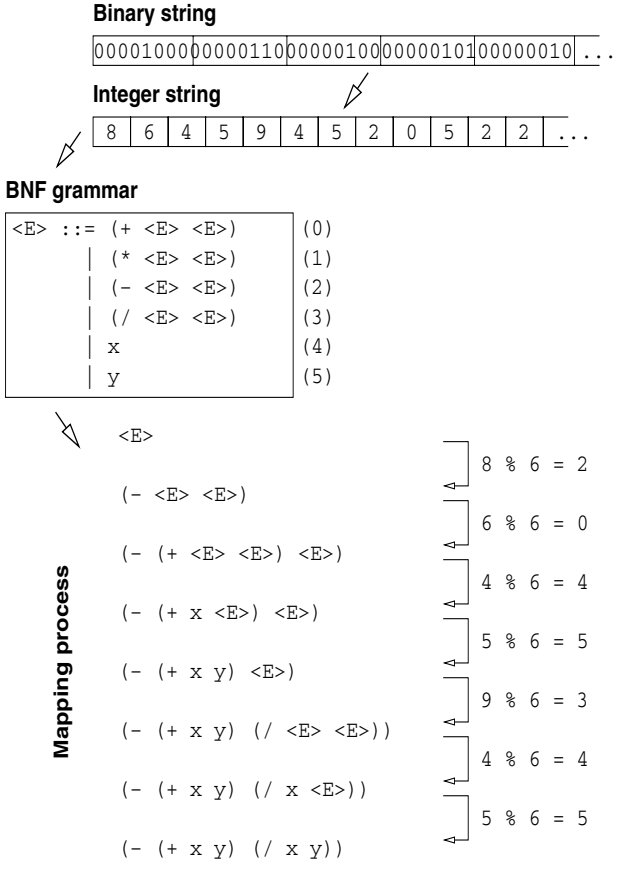

Fig. 2.2 The GE mapping process. A binary string is converted into an integer string, using 8 bits per codon. These integers are then used to choose productions from a BNF grammar, which map a given start symbol to a sequence of terminal symbols.

A variation on the termination of the mapping process is provided in Genr8 [87] where, upon reaching the end of the genotype wrapping occurs, but only production rules that result in terminating sequences are allowed to be selected. Effectively the grammar is modified to remove rules that expand the developing sentence.

2.2.1 πGE

A recent innovation in GE's mapping process has led to the development of Position Independent GE or πGE [157]. In standard GE, there is a positional dependency as the mapping process moves from left to right in consuming the non-terminals. πGE removes this dependency whereby instead of representing

each codon in standard GE as a single value, codons in πGE have two values: *nont* and *rule*. In this case, *nont* contains the encoding to select which non-terminal is to be consumed by the mapper. It does this using a rule similar to the original mapping rule:

$$NT = nont\%count$$

where NT is the non-terminal to be consumed (counted 0,1,...,n from left to right of the remaining non-terminals), *nont* is the value from the individual, and *count* is the number of non-terminals remaining. The rule part of the codon pair then, as in the previous section, selects which production rule to apply to the chosen non-terminal. For example, given that the grammar in the previous section produces the sequence `<exp><op><exp>` after the first symbol is consumed, these non-terminals would be counted `<exp>` – 0, `<op>` – 1 and `<exp>` – 2, giving 3 non-terminals. If the *nont* value was 8, this would yield an expression 8%3 selecting non-terminal 2 as the next to be consumed.

This more open-ended form of the mapping process was found to result in impressive performance gains on a number of benchmark problems.

2.3 Mutation and Crossover in GE

As already described, the representation of the individuals to which the genetic operators in GE are applied are variable-length linear strings. Due to this representation, the mechanics of the canonical GA operators are the same; mutation changes a bit or an integer to another random value, and one-point crossover swaps sections of the genetic code between parents. However, because of the mapping process, the effect on the phenotype can be complex.

When mutation is applied in standard GP, it occurs at the phenotypic level where the selected node or sub-tree is regenerated randomly and obeys the closure requirements of GP. In GE and Holland's GA, this operator is applied without constraints. However, in GE the potential exists for the mutation to have no effect, or to be *neutral*, at the phenotypic level [186]. For example, given the following BNF production rule:

```
<var>::= x              //(0)
       | y              //(1)
```

where the non-terminal `<var>` can be replaced with the variables `x` or `y`, and an integer codon value of 20. When `<var>` is consumed, it will be replaced with the variable `x` because, using the mapping rule from the previous section, the rule to be executed is 20 % 2; resulting in 0 or the first production rule. If a mutation occurs and changes the codon to 21, `<var>` will be replaced with `y`. However, if this codon is changed to 22, or any other even number for that matter, (as there are just two production rules) the phenotype will

remain as x. This means that neutral mutation is occurring as the resulting functionality of the expressed gene remains the same.

One-point crossover in GE is conducted under the same mechanics as its GA counterpart, where the genetic material to the right of the selected crossover points is swapped between two parents. In the case of GE, again due to the mapping process, the effect at the phenotypic level is different from that of GAs. Crossover in GE has a ripple effect on the derivation sequence after the crossover point. Figure 2.3 helps to illustrate this process. The mapping process in GE can be described as a sequence of derivations, which in turn can be represented as a derivation tree. At the core of crossover in GE

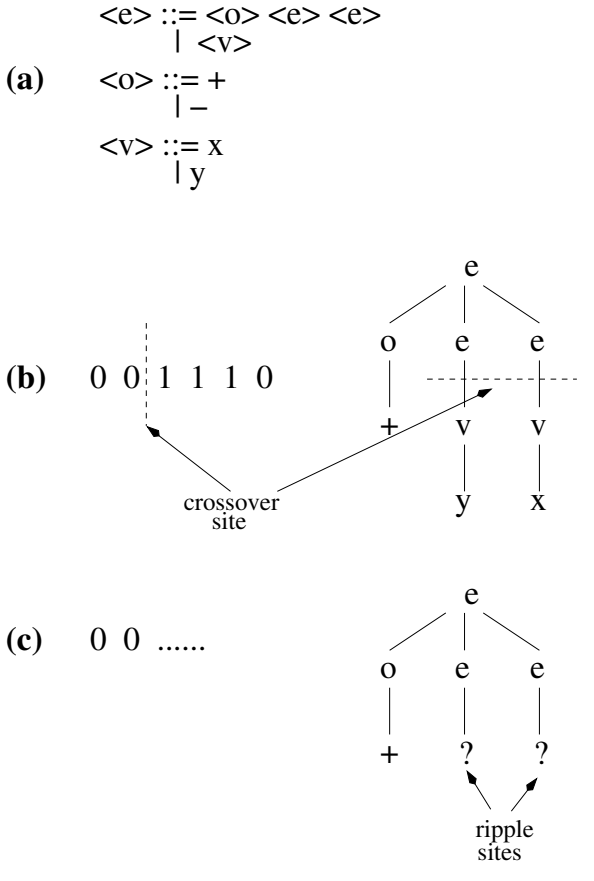

Fig. 2.3 Illustration of derivation tree during crossover. The BNF grammar (a) is applied to the genotype in (b) to produce the derivation tree on the right. A crossover site is selected after the first two bits with the resulting effect observed on the derivation tree in (c). The genotypic sequence which is added after the crossover point will be placed in a context at the ripple sites (denoted by the "?"s) which may differ from its original context.

is that when genetic material is swapped to another individual, the context in which it is placed may be different. This results in a different phenotypic expression compared to when it was in its original placement. While this form of crossover can appear to be destructive, research has shown that useful derivation sequences are transfered [155]. When analysing the behaviour of this *ripple crossover* it is clear that it exchanges multiple subtrees, and consequently significantly more material between solutions than standard sub-tree crossover, with 50% of an individual being exchanged on average. Further work by Harper & Blair [85] has also led to the development of a structure preserving two-point crossover that extracts information from the BNF grammar to minimise the destructive impact to useful blocks of code on either side of the crossover sub-section. This form of crossover is similar

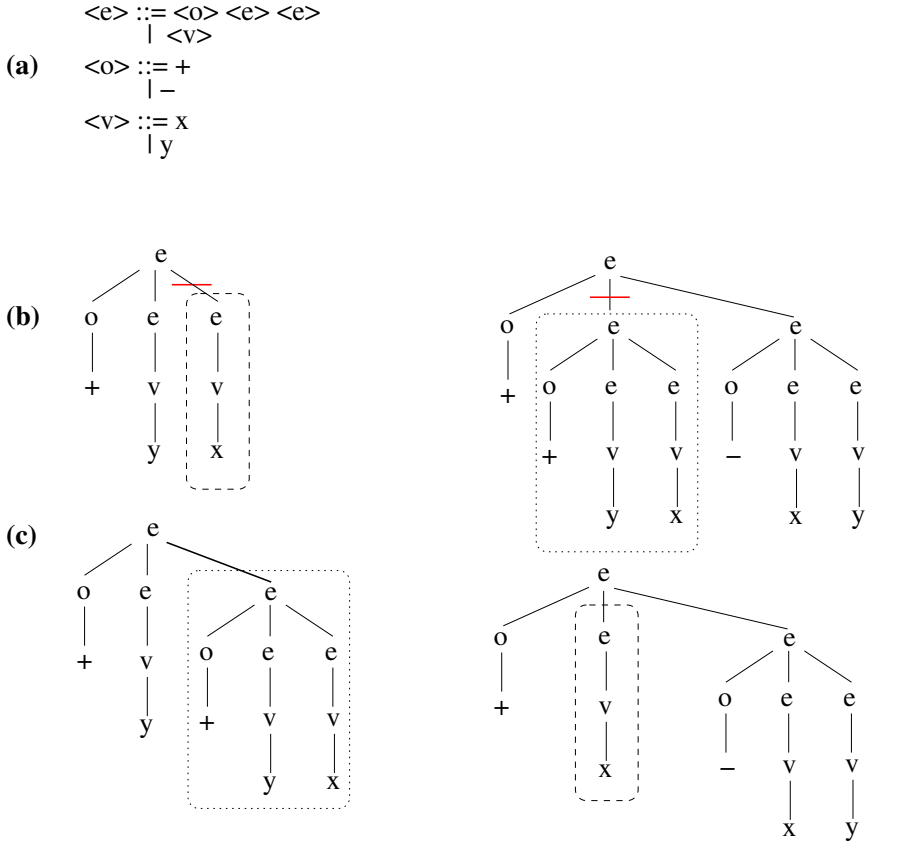

Fig. 2.4 Illustration of derivation tree during subtree crossover in GE. The BNF grammar (a) is applied to the genotype in (b) to produce the derivation tree on the right. A crossover site is selected after the first two bits with the resulting effect observed on the derivation tree in (c).

to sub-tree crossover in tree-based GP and an example of its operation illustrated in Figure 2.4.

Armed with more in-depth knowledge on the impact of GE's search operators to phenotypic change it is possible to design an appropriately balanced set of operators. Like any other form of EC it is essential that the genetic operators allow provision for the generation of both small and large changes, and to balance exploitation and exploration in the process. This is an active area of research within GE.

2.4 Alternative Search Engines

We hinted in the introduction to this chapter that it is possible to use alternative search strategies to the variable-length Evolutionary Algorithm of GE. Two prominent examples of this are evidenced by the use of Particle Swarm Optimisation and Differential Evolution to create the Grammatical Swarm (GS) [163] and Grammatical Differential Evolution (GDE) [162] variants. Note that we modify the second half of the algorithm's name to denote the search engine employed. We now provide a brief introduction to GS and GDE.

2.4.1 Grammatical Swarm

One model of social learning that has attracted interest in recent years is drawn from a swarm metaphor. Two popular variants of swarm models exist, those inspired by studies of social insects such as ant colonies, and those inspired by studies of the flocking behavior of birds and fish. The essence of these systems is that they exhibit flexibility, robustness and self-organisation [14]. Although the systems can exhibit remarkable coordination of activities between individuals, this coordination does not stem from a 'center of control' or a 'directed' intelligence, rather it is self-organising and emergent. 'Social Swarm' researchers have emphasized the role of social learning processes in these models [105, 106]. In essence, social behavior helps individuals to adapt to their environment, as it ensures that they obtain access to more information than that captured by their own senses.

In the context of PSO, a swarm can be defined as '... a population of interacting elements that is able to optimize some global objective through collaborative search of a space.' [105](p. xxvii). The nature of the interacting elements (particles) depends on the problem domain, in this study they represent program construction rules. These particles move (fly) in an n-dimensional search space, in an attempt to uncover ever-better solutions to the problem of interest.

Each of the particles has two associated properties, a current position and a velocity. Each particle has a memory of the best location in the search space that it has found so far (p_{best}), and knows the best location found to date by all the particles in the population (g_{best})(or in an alternative version of

the algorithm, a neighbourhood around each particle). At each step of the algorithm, particles are displaced from their current position by applying a velocity vector to them. The velocity size / direction is influenced by the velocity in the previous iteration of the algorithm (simulates 'momentum'), and the location of a particle relative to its p_{best} and g_{best}. Therefore, at each step, the size and direction of each particle's move is a function of its own history (experience), and the social influence of its peer group.

A number of variants of the particle swarm algorithm (PSA) exist. The following paragraphs provide a description of a basic *continuous* version of the algorithm.

i. Initialise each particle in the population by randomly selecting values for its location and velocity vectors.
ii. Calculate the fitness value of each particle. If the current fitness value for a particle is greater than the best fitness value found for the particle so far, then revise p_{best}.
iii. Determine the location of the particle with the highest fitness and revise g_{best} if necessary.
iv. For each particle, calculate its velocity according to equation 2.1.
v. Update the location of each particle according to equation 2.3.
vi. Repeat steps ii - v until stopping criteria are met.

The update algorithm for particle i's velocity vector v_i is:

$$v_i(t+1) = (w * v_i(t)) + (c_1 * R_1 * (p_{best} - x_i)) + (c_2 * R_2 * (g_{best} - x_i)) \quad (2.1)$$

where,

$$w = wmax - ((wmax - wmin)/itermax) * iter \quad (2.2)$$

In equation 2.1, p_{best} is the location of the best solution found to-date by particle i, g_{best} is the location of the global-best solution found by all particles to date, c_1 and c_2 are the weights associated with the p_{best} and the g_{best} terms in the velocity update equation, x_i is particle i's current location, and R_1 and R_2 are randomly drawn from U(0,1). The term w represents a momentum coefficient which is reduced according to equation 2.2 as the algorithm iterates. In equation 2.2, $itermax$ and $iter$ are the total number of iterations the algorithm will run for, and the current iteration value respectively; $wmax$ and $wmin$ set the upper and lower boundaries on the value of the momentum coefficient. The velocity update on any dimension is constrained to a maximum value of $vmax$. Once the velocity update for particle i is determined, its position is updated (equation 2.3, and p_{best} is updated if necessary (equations 2.4 & 2.5).

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (2.3)$$

$$y_i(t+1) = y_i(t) \text{ if, } f(x_i(t)) \leq f(y_i(t)) \quad (2.4)$$

$$y_i(t+1) = x_i(t) \text{ if, } f(x_i(t)) > f(y_i(t)) \tag{2.5}$$

After the location of all particles have been updated, a check is made to determine whether g_{best} needs to be updated (equation 2.6).

$$\hat{y} \in (y_0, y_1, ..., y_n) | f(\hat{y}) = \max(f(y_0), f(y_1), ..., f(y_n)) \tag{2.6}$$

In Grammatical Swarm (GS) the update equations for the swarm algorithm are as described earlier, with additional constraints placed on the velocity and particle location dimension values, such that maximum velocities *vmax* are bound to ±255, and each dimension is bound to the range [0,255] (denoted as *cmin* and *cmax* respectively). Note that this is a continuous swarm algorithm with real-valued particle vectors. The standard GE mapping function is adopted, with the real-values in the particle vectors being rounded up or down to the nearest integer value for the mapping process. In the current implementation of GS, fixed-length vectors are adopted, within which it is possible for a variable number of dimensions to be used during the program construction genotype-phenotype mapping process. A vector's elements (values) may be used more than once if wrapping occurs, and it is also possible that not all dimensions will be used during the mapping process if a complete program comprised only of terminal symbols, is generated before reaching the end of the vector. In this latter case, the extra dimension values are simply ignored and considered introns that may be switched on in subsequent iterations.

GS Experimental Findings

A diverse selection of benchmark programs from the literature were tackled to demonstrate proof of concept for the GS method. The problems included Santa Fe Ant Trail, a Symbolic Regression instance ($x + x^2 + x^3 + x^4$), the 3-Multiplexer boolean problem, and Mastermind. The parameters adopted across the experiments were $c_1 = c_2 = 1.0$, $wmax = 0.9$, $wmin = 0.4$, $cmin = 0$ (minimum value a coordinate may take), $cmax = 255$ (maximum value a coordinate may take). In addition, a swarm size of 30 running for 1000 iterations using 100 dimensions is used. The same problems are also tackled with GE in order to determine how well GS is performing at program generation in relation to the more traditional variable-length Genetic Algorithm search engine of standard GE. In an attempt to achieve a relatively fair comparison of results given the differences between the search engines of Grammatical Swarm and Grammatical Evolution, we have restricted each algorithm in the number of individuals they process. Grammatical Swarm running for 1000 iterations with a swarm size of 30 processes 30,000 individuals, therefore, a standard population size of 500 running for 60 generations is adopted for Grammatical Evolution. The remaining parameters for Grammatical

Evolution are roulette selection, steady state replacement, one-point crossover with probability of 0.9, and a bit mutation with probability of 0.01.

Table 2.1 provides a summary and comparison of the performance of GS and GE on each of the problem domains tackled. 100 independent runs were performed for the data reported. In two out of the four problems GE outperforms GS, and GS outperforms GE on the other two problem instances. The key finding is that the results demonstrate proof of concept that GS can successfully generate solutions to problems of interest. In this initial study, we have not attempted parameter optimisation for either algorithm, but results and observations of the particle swarm engine suggests that swarm diversity is open to improvement. We note that a number of strategies have been suggested in the swarm literature to improve diversity [202], and we suspect that a significant improvement in GS performance can be obtained with the adoption of these measures. Given the relative simplicity of the Swarm algorithm, the small population sizes involved, and the complete absence of a crossover operator synonymous with program evolution in GP, it is impressive that solutions to each of the benchmark problems have been obtained.

Table 2.1 A comparison of the results obtained for Grammatical Swarm and Grammatical Evolution across all the problems analysed

	Mean Best Fitness (Std.Dev.)	**Mean Average Fitness (Std.Dev.)**	**Successful Runs**
Santa Fe ant			
GS	75.24 (16.64)	33.43 (3.69)	43
GE	80.18 (13.79)	46.43 (11.18)	58
Multiplexer			
GS	0.97 (0.05)	0.87 (0.01)	79
GE	0.95 (0.06)	0.88 (0.04)	56
Symbolic Regression			
GS	0.31 (0.35)	0.07 (0.02)	20
GE	0.88 (0.30)	0.28 (0.28)	85
Mastermind			
GS	0.91 (0.04)	0.88 (0.01)	18
GE	0.90 (0.03)	0.89 (0.00)	10

2.4.2 Grammatical Differential Evolution

Differential evolution (DE) [209, 210, 211, 175] is a population-based search algorithm. The algorithm draws inspiration from the field of Evolutionary Computation, as it embeds implicit concepts of mutation, recombination and fitness-based selection, to evolve from an initial randomly generated population to a solution to a problem of interest. It also borrows principles from

Social Algorithms through the manner in which new individuals are generated. Unlike the binary chromosomes typical of GAs, an individual in DE is generally comprised of a real-valued chromosome.

Although several DE algorithms exist we only describe one version of the algorithm based on the *DE/rand/1/bin* scheme [209]. The different variants of the DE algorithm are described using the shorthand DE/x/y/z, where x specifies how the base vector to be perturbed is chosen (*rand* if it is randomly selected or *best* if the best individual is selected), y is the number of difference vectors used, and z denotes the crossover scheme used (*bin* for crossover based on independent bi-nominal experiments, and *exp* for exponential crossover).

At the start of this algorithm, a population of N, d-dimensional vectors $X_j = (x_{i1}, x_{i2}, \ldots, x_{id})$, $j = 1, \ldots, n$, is randomly initialised and evaluated using a fitness function f. During the search process, each individual (j) is iteratively refined. The modification process has three steps:

i. Create a variant solution, using randomly selected members of the population.
ii. Create a trial solution, by combining the variant solution with j (crossover step).
iii. Perform a selection process to determine whether the trial solution replaces j in the population.

Under the mutation operator, for each vector $X_j(t)$, a variant solution $V_j(t+1)$ is obtained using equation 2.7:

$$V_j(t+1) = X_m(t) + F(X_k(t) - X_l(t)) \tag{2.7}$$

where $k, l, m \in 1, \ldots, N$ are mutually different, randomly selected indices, and all the indices $\neq j$ (X_m is referred to as the base vector, and $X_k(t) - X_l(t)$ is referred to as a difference vector). Variants on this step include the use of more than three individuals from the population, and/or the inclusion of the highest-fitness point in the population as one of these individuals [209]. The difference between vectors X_k and X_l is multiplied by a scaling parameter F (typically, $F \in (0, 2]$). The scaling factor controls the amplification of the difference between X_k and X_l, and is used to avoid stagnation of the search process. Following the creation of the variant solution, a trial solution $U_j(t+1) = (u_{j1}, u_{j2}, \ldots, u_{jd})$ is obtained from equation 2.8.

$$U_{jn}(t+1) = \begin{cases} V_{jn}, \text{if } (rand \leq CR) \text{ or } (j = rnbr(i)) \ ; \\ X_{jn}, \text{if } (rand > CR) \text{ and } (j \neq rnbr(i)). \end{cases} \tag{2.8}$$

where $n = 1, 2, \ldots, d$, *rand* is drawn from a uniform random number generator in the range (0,1), CR is the user-specified crossover constant from the range (0,1), and $rnbr(i)$ is a randomly chosen index chosen from the range $(1, 2, \ldots, n)$. The random index is used to ensure that the trial solution differs by at least one component from $X_i(t)$. The resulting trial solution replaces its

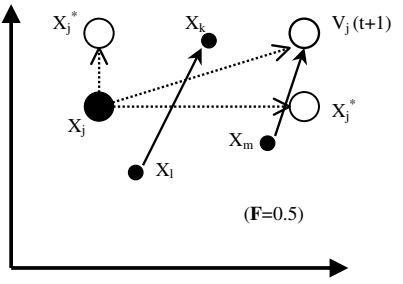

Fig. 2.5 A representation of the Differential Evolution variety-generation process. The value of F is set at 0.50. In a simple 2-d case, the child of particle X_j can end up in any of three positions. It may end up at either of the two positions X_j^*, or at the position of particle $V_j(t+1)$.

predecessor , if it has higher fitness (a form of selection), otherwise the predecessor survives unchanged into the next iteration of the algorithm (equation 2.9).

$$X_i(t+1) = \begin{cases} U_i(t+1), & \text{if } f(U_i(t+1)) < f(X_i(t)); \\ X_i(t), & \text{otherwise.} \end{cases} \tag{2.9}$$

Fig. 2.5 provides a graphic of the adaptive process of GDE. The DE algorithm has three parameters, the population size (N), the crossover rate (CR), and the scaling factor (F). Higher values of CR tend to produce faster convergence of the population of solutions. Typical values for these parameters are in the range, N=50-100 (or ten times the number of dimensions in a solution vector), CR=0.8-0.9 and F=0.3-0.5.

Grammatical Differential Evolution (GDE) adopts a Differential Evolution learning algorithm coupled to a Grammatical Evolution (GE) genotype-phenotype mapping to generate programs in an arbitrary language. The standard GE mapping function is adopted with the real-values in the vectors being rounded up or down to the nearest integer value, for the mapping process. In the current implementation of GDE, fixed-length vectors are adopted within which it is possible for a variable number of elements to be required during the program construction genotype-phenotype mapping process. A vector's values may be used more than once if the wrapping operator is used, and in the opposite case it is possible that not all elements will be used during the mapping process if a complete program comprised only of terminal symbols is generated before reaching the end of the vector. In this latter case, the extra element values are simply ignored and considered introns that may be switched on in subsequent iterations.

GDE Experimental Findings

The same diverse set of problems are tackled with GDE as with GS, including an instance of Symbolic Regression ($x+x^2+x^3+x^4$), the Santa Fe Ant Trail,

boolean 3-Multiplexer, and Mastermind. The parameters adopted across the following experiments are Params of GDE....popsize 500, 100 iterations, strlen 100, F=0.9, CR=1.0, DE/best/1/exp. Gene values are bound to the range $[0 \rightarrow 255]$.

The same problems are also tackled with Grammatical Evolution in order to get some indication of how well GDE is performing at program generation in relation to the more traditional variable-length Genetic Algorithm-driven search engine of standard GE. a standard population size of 500 running for 60 generations is adopted for Grammatical Evolution. The remaining parameters for Grammatical Evolution are roulette selection, steady state replacement, one-point crossover with probability of 0.9, and a bit mutation with probability of 0.01.

Table 2.2 provides a summary and comparison of the performance of Grammatical Differential Evolution, and Grammatical Evolution on each of the problem domains tackled. The reported results are averaged over 50 independent runs. In three out of the four problems Grammatical Evolution outperforms GDE. The key finding is that the results demonstrate proof of concept that GDE can successfully generate solutions to problems of interest.

Table 2.2 A comparison of the results obtained for Grammatical Differential Evolution and Grammatical Evolution across all the problems analysed

	Santa Fe Ant	**Symbolic Multiplexer**	**Regression**	**Mastermind**
GDE/rand/1/bin	10	23	6	0
GDE/best/1/exp	7	**27**	4	0
GDE/rand-to-best/1/exp	9	**27**	4	0
GDE/rand-to-best/1/bin	7	25	5	0
GE	**17**	15	**24**	**3**

2.5 Applications of GE

Since its inception GE has received considerable attention and been applied to a wide variety of problem domains. Early studies saw GE being successfully applied to symbolic-regression problems [190], the evolution of trigonometric identities [192], the evolution of caching algorithms [145], and behavioural robotics [146, 147].

GE was extended and applied to the domain of surface design in [87] where it was combined with GENR8, a surface design tool that uses an evolutionary process to adapt surfaces to meet a user's specification. In the field of Bioinformatics, GE was applied to recognise eukaryotic promotors [161] that help in the identification of biological genes. While in [137] GE was used to evolve neural networks for feature selection in genetic epidemiology.

In the area of sound synthesis and analysis Ortega et al. [168] use GE to automatically generate compositions which were comparable to human

composed works. In [109] GE was tasked with evolving phonological rules which can be used to translate text into graphemes and recognition of sounds as words.

Fractal curves of a high dimensionality were evolved in [169] and in [130] Petri-Net models of complex systems were evolved. GE has also been used in the design Logos [164].

In the financial domain, GE has been applied in a number of areas [24], with studies in foreign-exchange trading [17, 18, 22], bankruptcy and corporate analysis [16, 19, 20], and credit classification [21, 23]. Specific to the trading of market indices, which is examined in this book, studies have also been conducted [149, 150, 15] where the problem is examined in a static manner.

2.6 Conclusion

This chapter introduced Grammatical Evolution and described its mapping process and how the genetic operators of mutation and crossover are implemented along with their effects. Some noteable applications of GE were outlined.

In addition to the various papers on the subject (see the GP Bibliography [115]), further information on GE can be found from `http://www.grammatical-evolution.org` including links to various software implementations of GE. A version of GE in Java, GEVA [76], has recently been released by our Natural Computing Research & Applications group at University College Dublin. This is available directly from `http://ncra.ucd.ie/geva/`.

Subsequent chapters will explore the features of GE and its potential for use in dynamic environments. Before we begin a more in-depth analysis of GE in dynamic environments, in the following chapter we first provide an overview of the research to date on Evolutionary Computation in these non-stationary domains.