Natural Computing (2006) 5: 443–462
DOI 10.1007/s11047-006-9007-7

Grammatical Swarm: The generation of programs by social programming

MICHAEL O'NEILL* and ANTHONY BRABAZON
*Natural Computing Research & Applications Group, University College Dublin, Dublin, Ireland (*Author for correspondence, e-mail: m.oneill@ucd.ie)*

Abstract. This study examines Social Programming, that is, the construction of programs using a Social Swarm algorithm based on Particle Swarm Optimization. Each individual particle represents choices of program construction rules, where these rules are specified using a Backus–Naur Form grammar. This study represents the first instance of a Particle Swarm Algorithm being used to generate programs. A selection of benchmark problems from the field of Genetic Programming are tackled and performance is compared to Grammatical Evolution. The results demonstrate that it is possible to successfully generate programs using the Grammatical Swarm technique. An analysis of the Grammatical Swarm approach is presented on the dynamics of the search. It is found that restricting the search to the generation of complete programs, or with the use of a ratchet constraint forcing individuals to move only if a fitness improvement has been found, can have detrimental consequences for the swarms performance and dynamics.

Key words: genetic programming, grammatical evolution, particle swarm optimization, social learning, social programming

1. Introduction

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. This study focuses on the latter. The essence of these systems is that they exhibit flexibility, robustness and self-organization (Bonabeau et al., 1999). 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-organizing and emergent. Social swarm researchers have emphasized the role of social learning processes in these models (Kennedy and Eberhart, 1995; Kennedy et al., 2001). 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.

This paper details an investigation examining the possibility of specifying the automated construction of a program using a Particle Swarm learning model. The results demonstrate that the performance of this approach on a number of benchmark problems is comparable with, and in some cases superior to, a popular grammar-based form of Genetic Programming, namely Grammatical Evolution. In the Grammatical Swarm (GS) methodology developed in this study, each particle or realvalued vector, represents choices of program construction rules specified as production rules of a Backus–Naur Form grammar.

GS is grounded in the linear Genetic Programming representation adopted in Grammatical Evolution (GE) (O'Neill and Ryan, 2003), which uses grammars to guide the construction of syntactically correct programs, specified by variable-length genotypic binary or integer strings.

There are a number of advantages of a grammatical approach to genetic programming, including the ability to encode multiple data types into the solutions generated. In tree-based GP an individual is restricted to a single type (typically real values) due to the restriction of closure. That is, every operator within the tree must be capable of handling all possible inputs from its subtrees. The non-terminal symbols within a grammatical approach effectively allow the developing program to contain multiple data types.

Due to the genotype–phenotype mapping involved in GE (and GS) it is possible to conveniently encode domain knowledge into the grammar which can be used to bias the construction of solutions. Adopting a grammatical representation coupled to the genotype–phenotype mapping allows the user to easily change the language generated by simply modifying the grammar thus making such an approach language independent. Also due to the genotype–phenotype separation it is possible to change the search heuristic that operates on the binary or integer strings that represent the genotype. The search heuristic adopted with GE is a variable-length Genetic Algorithm. In the GS technique presented here, a particle's real-valued vector is used in the same manner as the genotypic binary string in GE. This results in a new form of automatic programming based on social learning, which we dub Social Programming, or *Swarm*

Programming. It is interesting to note that this approach is completely devoid of any crossover operator.

The remainder of the paper is structured as follows. Before describing the Grammatical Swarm algorithm in Section 4, introductions to the salient features of Particle Swarm Optimization (PSO) and Grammatical Evolution (GE) are provided in Sections 2 and 3, respectively. Section 5 details the experimental approach adopted and results, Section 6 contains an analysis of various methods of initialization and update strategies in GS, and finally Section 7 details conclusions and opportunities for future work.

2. Particle Swarm optimization

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.' (Kennedy et al., 2001, 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 (or in an alternative version of the algorithm, a neighborhood around each particle) (g_{best}). 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. Initialize each particle in the population by randomly selecting values for its location and velocity vectors.

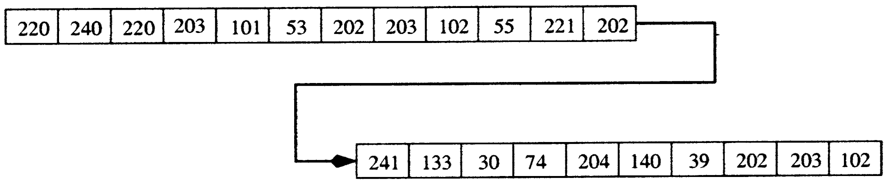


Figure 1. An example of a genome in GE, represented as integers for ease of reading.

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 1.
v. Update the location of each particle according to Equation 3.
vi. Repeat steps ii–v until stopping criteria are met.

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

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

where,

$$w = wmax - ((wmax - wmin)/itermax) * iter \tag{2}$$

In Equation 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). *w* represents a momentum coefficient which is reduced according to Equation 2 as the algorithm iterates. In Equation 2, *itermax* and *iter* are the total number of iterations the algorithm will run for, and the current iteration value, respectively, and *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 3), and p_{best} is updated if necessary (Equations 4 and 5).

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{3}$$

$$y_i(t+1) = y_i(t) \quad \text{if } f(x_i(t)) \leq f(y_i(t)) \tag{4}$$

$$y_i(t+1) = x_i(t) \quad \text{if } f(x_i(t)) > f(y_i(t)) \tag{5}$$

After the location of all particles have been updated, a check is made to determine whether g_{best} needs to be updated (Equation 6).

$$\hat{y} \in (y_0, y_1, \ldots, y_n) |\, f(\hat{y}) = \max\,(f(y_0), f(y_1), \ldots, f(y_n)) \tag{6}$$

3. Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary algorithm that can evolve computer programs in any language (Ryan et al., 1998; O'Neill, 2001; O'Neill and Ryan, 2001, 2003; O'Neill et al., 2003), and represents a grammar-based genetic programming approach. Rather than representing the programs as parse trees, as in GP (Koza, 1992, 1994; Banzhaf et al., 1998; Koza et al., 1999, 2003), a linear genome representation is used (Figure 1). A genotype–phenotype mapping is employed such that each individual's variable length binary string, contains in its codons (groups of 8 bits) the information to select production rules from a Backus–Naur Form (BNF) grammar (see Figure 2). Consequently, the genetic operators such as crossover and mutation are applied to the linear genotype in a typical genetic algorithm manner, unlike in a tree-based Genetic Programming approach where they are applied directly to the phenotypic parse trees. The grammar allows the generation of programs in an arbitrary language that are guaranteed to be syntactically correct, and as such it is used as a generative grammar, as opposed to the classical use of grammars in compilers to check syntactic correctness of sentences. The user can tailor the grammar to produce solutions that are purely syntactically constrained, or they may incorporate domain knowledge by biasing the grammar to produce very specific forms of sentences.

BNF is a notation that represents a language in the form of production rules. It is comprised of a set of non-terminals that can be mapped to elements of the set of terminals (the primitive symbols that can be used to construct the output program), according to the production rules. A simple example BNF grammar is given below, where

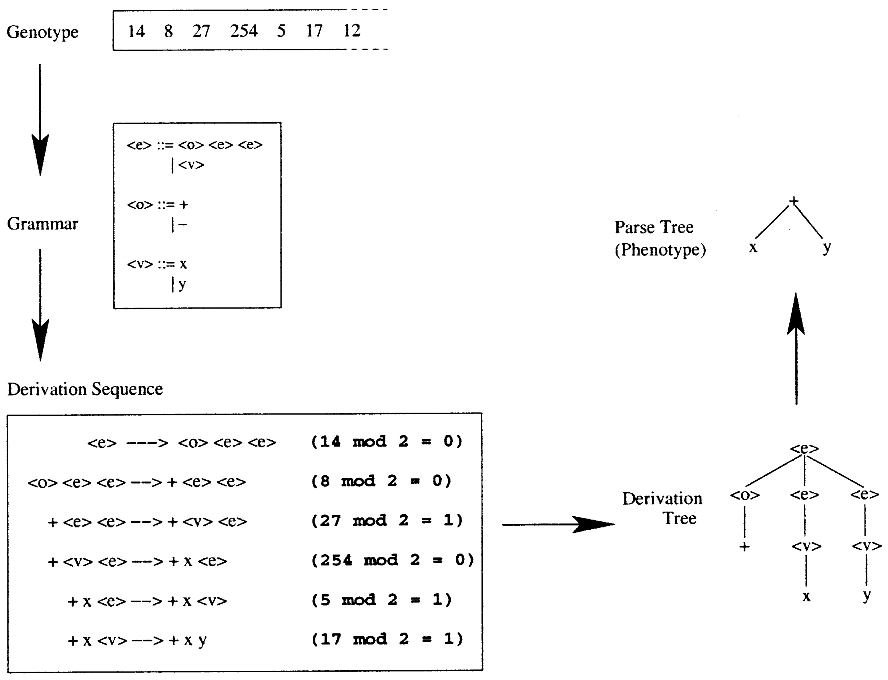

Genotype
14 8 27 254 5 17 12
Grammar
<e> ::= <o> <e> <e>
| <v>
<o> ::= +
| -
<v> ::= x
| y
Derivation Sequence
<e> ---> <o> <e> <e> (14 mod 2 = 0)
<o> <e> <e> ---> + <e> <e> (8 mod 2 = 0)
+ <e> <e> ---> + <v> <e> (27 mod 2 = 1)
+ <v> <e> ---> + x <e> (254 mod 2 = 0)
+ x <e> ---> + x <v> (5 mod 2 = 1)
+ x <v> ---> + x y (17 mod 2 = 1)
Derivation Tree
Parse Tree (Phenotype)
<e> <o> <e> <e> + <v> <v> x y
+ x y

Figure 2. An example of a GE genotype–phenotype mapping, where the genotype is used to select production rules from a grammar to produce a derivation sequence. The derivation sequence represents the development of a program from the embryonic non-terminal start symbol (< e >). The derivation sequence can be represented as a derivation tree, which can then be simplified to correspond to the parse tree adopted in standard tree-based Genetic Programming.

< expr > is the start symbol from which all programs are generated. These productions state that < expr > can be replaced with either one of < expr > < op > < expr > or < var >. An < op > can become either +, −, or *, and a < var > can become either *x*, or *y*.

```
<expr>  ::=  <expr><op><expr>   (0)
          |  <var>              (1)
<op>    ::=  +                  (0)
          |  -                  (1)
          |  *                  (2)
<var>   ::=  x                  (0)
          |  y                  (1)
```

The grammar is used in a developmental process to construct a program by applying production rules, selected by the genome, beginning from the start symbol of the grammar. In order to select a production rule in GE, the next codon value on the genome is read, interpreted, and placed in the following formula:

$$Rule = c \,\%\, r$$

where c represents the current codon value, % represents the modulus operator, and r is the number of choices for the current non-terminal.

Given the example individual's genome (where each 8-bit codon is represented as an integer for ease of reading) in Figure 2, the first codon integer value is 220, and given that we have 2 rules to select from for `<expr>` as in the above example, we get 220 % 2 = 0. `<expr>` will therefore be replaced with `<expr><op><expr>`.

Beginning from the left hand side of the genome, codon integer values are generated and used to select appropriate rules for the left most non-terminal in the developing program from the BNF grammar, until one of the following situations arise: (a) A complete program is generated. This occurs when all the non-terminals in the expression being mapped are transformed into elements from the terminal set of the BNF grammar. (b) The end of the genome is reached, in which case the *wrapping* operator 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 unless an upper threshold representing the maximum number of wrapping events has occurred during this individuals mapping process. (c) In the event that a threshold on the number of wrapping events has occurred and the individual is still incompletely mapped, the mapping process is halted, and the individual assigned the lowest possible fitness value. Returning to the example individual, the left-most `<expr>` in `<expr><op><expr>` is mapped by reading the next codon integer value 240 and used in 240 % 2 = 0 to become another `<expr><op><expr>`. The developing program now looks like `<expr><op><expr><op><expr>`. Continuing to read subsequent codons and always mapping the left-most non-terminal the individual finally generates the expression $y * x-x-x+x$, leaving a number of unused codons at the end of the individual, which are deemed to be introns and simply ignored. A full description of GE can be found in O'Neill and Ryan (2003), and some more recent applications and extensions including the use of meta-grammars

(Grammatical Evolution by Grammatical Evolution and the meta-Grammar Genetic Algorithm) and an alternative approach to the mapping process (πGE) can be found in Brabazon and O'Neill (2006).

4. Grammatical Swarm

Grammatical Swarm (GS) adopts a Particle Swarm algorithm coupled to a Grammatical Evolution (GE) genotype–phenotype mapping to generate programs in an arbitrary language (O'Neill and Brabazon, 2004). The update equations for the particle 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 is generated before reaching the end of the vector. In this latter case, the extra dimension values are simply ignored and can be considered as introns that may be switched on in subsequent iterations.

5. Proof of concept experiments and results

A diverse selection of benchmark programs from the literature on evolutionary automatic programming are tackled using Grammatical Swarm to demonstrate proof of concept for the GS methodology. The parameters adopted across the following experiments are $c_1 = c_2 = 1.0$, $wmax = 0.9$, $wmin = 0.4$, $cmin = 0$ (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.

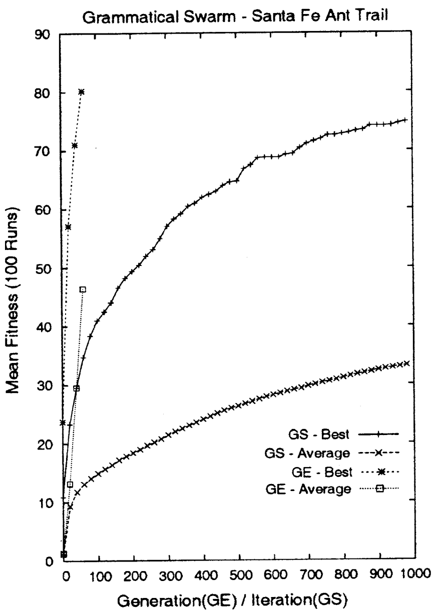
The same problems are also tackled with GE in order to determine how well GS is performing at program generation relative to the more traditional variable-length Genetic Algorithm search engine of standard GE. In an attempt to achieve a relatively fair comparison of results given the differences between 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.

5.1. *Santa Fe ant trail*

The Santa Fe ant trail is a standard problem in the area of GP and can be considered a deceptive planning problem with many local and global optima (Langdon and Poli, 1998). The objective is to find a computer program to control an artificial ant so that it can find all 89 pieces of food located on a non-continuous trail within a specified number of time steps, the trail being located on a 32 by 32 toroidal grid. The ant can only turn left, right, move one square forward, and may also look ahead one square in the direction it is facing to determine if that square contains a piece of food. All actions, with the exception of looking ahead for food, take one time step to execute. The ant starts in the top left-hand corner of the grid facing the first piece of food on the trail. The grammar used in this experiment is different to the ones used later for symbolic regression and the multiplexer problems in that we wish to produce a multi-line function in this case, as opposed to a single line expression. The grammar for the Santa Fe ant trail problem is given below.

```
<code> ::= <line> | <code> <line>
<line> ::= <condition> | <op>
<condition> ::=if(food_ahead()) { <line> }else
                { <line> }
<op> ::=left();|right();|move();
```

A plot of the mean best fitness and cumulative frequency of success for 100 runs can be seen in Figure 3. As can be seen, convergence towards the best fitness occurs, and a number of runs successfully obtain the correct solution (best fitness is achieved when all 89 pieces of food are eaten).



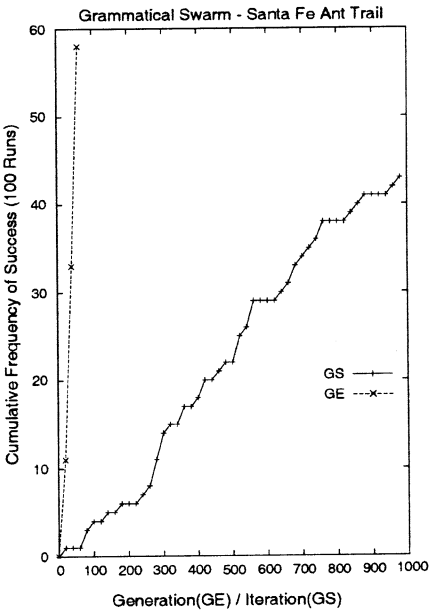


Figure 3. Plot of the mean fitness on the Santa Fe ant Trail problem instance (left), and the cumulative frequency of success (right).

5.2. *Quartic symbolic regression*

The target function is defined as $f(a) = a + a^2 + a^3 + a^4$, and 100 randomly generated input–output vectors are created for each call to the target function, with values for the input variable drawn from the range [0,1]. The fitness for this problem is given by the reciprocal of the sum, taken over the 100 fitness cases, of the absolute error between the evolved and target functions. The grammar adopted for this problem is as follows:

```
<expr> ::= <expr> <op> <expr> | <var>
<op> ::=  + | - | * | /
<var> ::=a
```

A plot of the cumulative frequency of success and the mean best fitness over 100 runs can be seen in Figure 4. As can be seen, a number of runs successfully find the correct solution to the problem. GE clearly outperforms GS on this instance.

5.3. *3 Multiplexer*

An instance of a multiplexer problem is tackled in order to further verify that it is possible to generate programs using Grammatical Swarm. The aim with this problem is to discover a Boolean expression that behaves as a 3 Multiplexer. There are eight fitness cases for this instance, representing all possible input–output pairs. Fitness is the number of input cases for which the evolved expression returns the correct output. The grammar adopted for this problem is as follows:

```
<mult> ::= guess = <bexpr>;
<bexpr> ::= (<bexpr> <bilop> <bexpr>)
          |<ulop> (<bexpr>)
          |<input>
<bilop> ::= and |or
<ulop> ::= not
<input> ::= input0 | input1 | input2
```

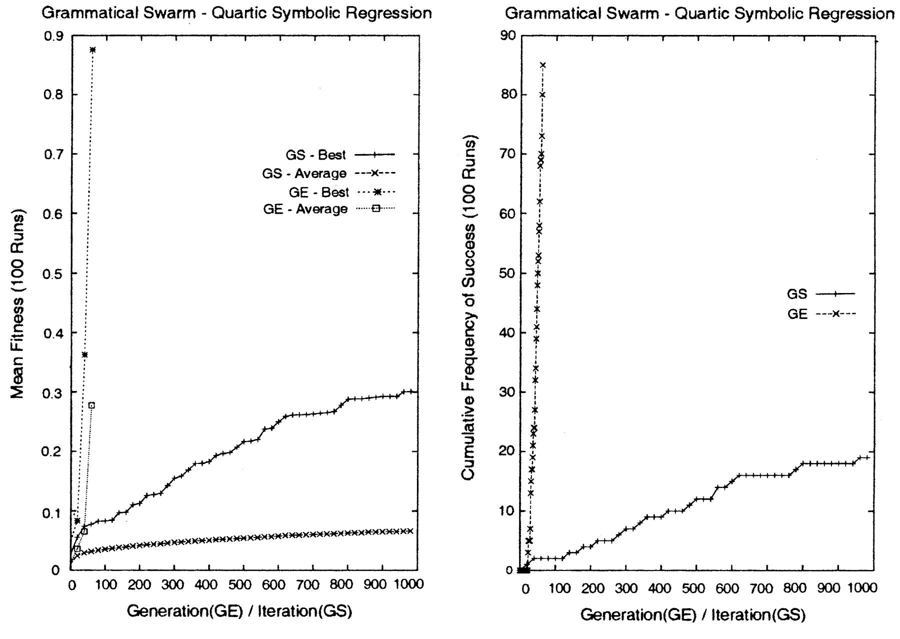


Figure 4. Plot of the mean fitness on the quartic symbolic regression problem instance (left), and the cumulative frequency of success (right).

A plot of the mean best fitness over 100 runs can be seen in Figure 5. As can be seen, convergence towards the best fitness occurs, and a number of runs successfully evolve correct solutions. In this instance GS is outperforming GE.

5.4. *Mastermind*

In this problem, the code breaker attempts to guess the correct combination of colored pins in a solution. When an evolved solution to this problem (i.e. a combination of pins) is to be evaluated, it receives one point for each pin that has the correct color, regardless of its position. If all pins are in the correct order then an additional point is awarded to that solution. This means that ordering information is only presented when the correct order has been found for the whole string of pins.

A solution therefore, is in a local optimum if it has all the correct colors, but in the wrong positions. The difficulty of this problem is controlled by the number of pins and the number of colors in the

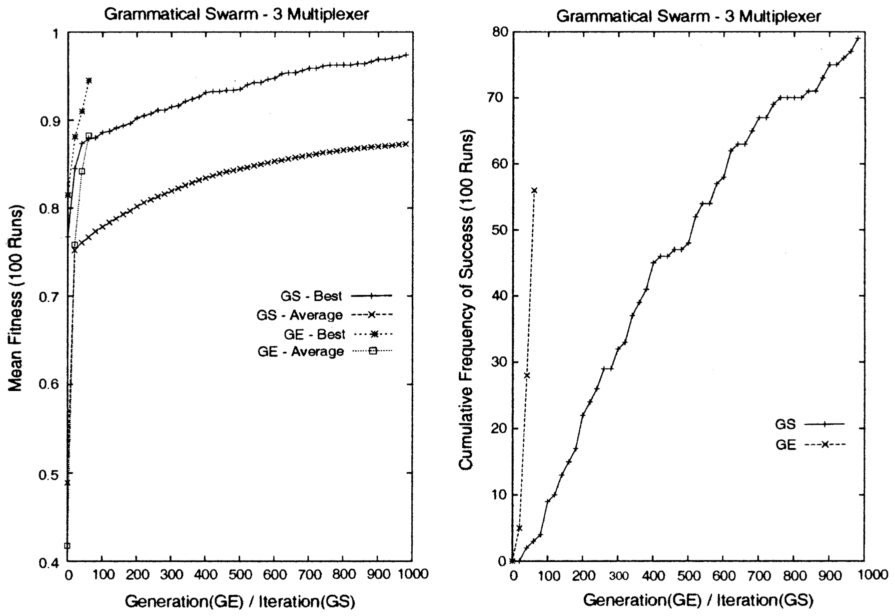


Figure 5. Plot of the mean fitness on the 3 multiplexer problem instance (left), and the cumulative frequency of success (right).

target combination. The instance tackled here uses four colors and eight pins with the following target values 3 2 1 3 1 3 2 0.

Results are provided in Figure 6 and the grammar adopted is as follows.

```
<pin> ::= <pin> <pin> | 0 | 1 | 2 | 3
```

5.5. *Summary*

Table 1 provides a summary and comparison of the performance of GS and GE on each of the problem domains tackled. To ensure a fair comparison of both approaches, typical population sizes for each algorithm are adopted and then the same number of individuals are processed for each algorithm by fixing the number of generations in each case. Again, the number of generations for each algorithm is typical of settings found in the literature for both GE and PSO. We

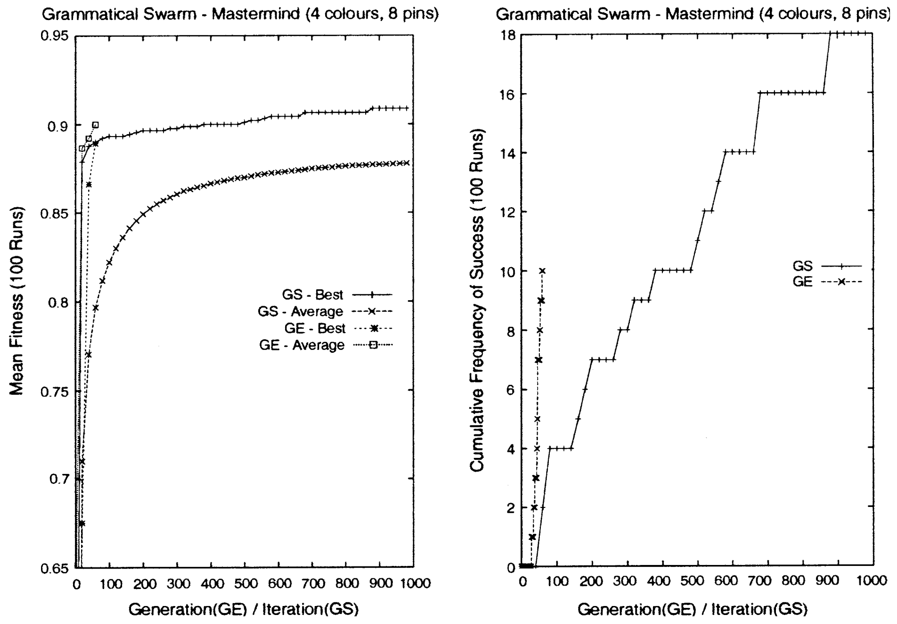


Figure 6. Plot of the mean best and mean average fitness (left) and the cumulative frequency of success (right) on the Mastermind problem instance using eight pins and four colors. Fitness is defined as the points score of a solution divided by the maximum possible points score.

Table 1. A comparison of the results obtained for Grammatical Swarm and Grammatical Evolution across all the problems analyzed averaged over 100 runs

	Mean best fitness	SD	Median	Runs successful
Santa Fe ant				
GS	75.24	16.64	88.00	43
GE	**80.18**	**13.79**	**89.00**	**58**
Multiplexer				
GS	**0.97**	**0.05**	**1.00**	**79**
GE	0.95	0.06	1.00	56
Symbolic regression				
GS	0.31	0.35	0.16	20
GE	**0.88**	**0.30**	**1.0**	**85**
Mastermind				
GS	0.91	0.04	0.89	18
GE	0.90	0.03	0.89	10

A *t*-test and bootstrap (re-sampling) *t*-test were performed at the 95% confidence level on the best fitness values, and where one algorithm outperforms another on the best fitness is highlighted in bold.

measure the success of the GS approach based on the number of times a successful solution is found out of the 100 runs conducted for each problem. In two out of the four problems GE outperforms GS in terms of the number of correct solutions found, and GS outperforms GE on the other two problem instances. Specifically, 79% of GS runs were successful in producing a correct solution on the Multiplexer problem versus 56% for GE, with GS solving 18% of the time on the Mastermind problem versus GEs 10%. Additionally, a *t*-test and bootstrap (re-sampling) *t*-test was performed on the best fitness values at the 95% confidence level. GE was found to outperform GS on the Santa Fe ant and Symbolic Regression problem instances. GS outperformed GE on the Multiplexer instance, and there was no difference between GE and GS on the Mastermind instance. The key finding is that the results demonstrate proof of concept that GS can successfully generate solutions to problems of interest, and represents the first time that a Particle Swarm algorithm has been used to generate programs. In this initial study, we have not attempted parameter optimization for either algorithm. We note that a number of strategies have been suggested in the swarm literature to improve diversity (Silva et al., 2002), and it is likely 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.

6. Analysis of initialization and update constraints

In this set of experiments, a set of initialization and update constraints are examined. Performance of the standard GS algorithm is compared to a setup where (1) all individuals in the initial population are forced to be a valid program (that is an executable program that is completely mapped to terminal symbols), (2) when undergoing the update step, a particle can only move to next point in *n*-dimensional space when the new position is a valid program, and (3) a ratchet strategy is examined that constrains the update of particles by restricting their movement to positions of higher fitness. Combinations of these three setups are also tested for completeness.

The first two of these experiments test the hypothesis that forcing individuals to map to complete executable programs has no effect on the performance of the GS algorithm when compared to the setup where individuals may produce an incompletely mapped individual that cannot be executed. The first experiment examines this issue at the initialization step, and the second experiment studies the update step. The third experiment tests the hypothesis that there is no difference between allowing a swarm to fly through its search space unconstrained by an individual's fitness, compared to the case where only fitness improvements result in a particle's movement to a new position. In a study on a particle swarm model of Organizational Adaptation it has been demonstrated that restricting the movement of particles in this manner can improve an organization's adaptive ability (Brabazon et al., 2005). We wish to test if this is also the case for GS.

The benchmark problem instances adopted earlier are tackled for all of the analysis experiments. The cumulative frequency of success for each problem domain are presented in Figures 7 and 8, and a summary of the results including the mean average and mean best fitness values at the end of the runs are presented in Table 2.

Across all problems the Ratchet strategy on its own and in combination with Valid Initialization underperforms relative to all the other setups. Based on a *t*-test and bootstrap (re-sampling) *t*-test comparing

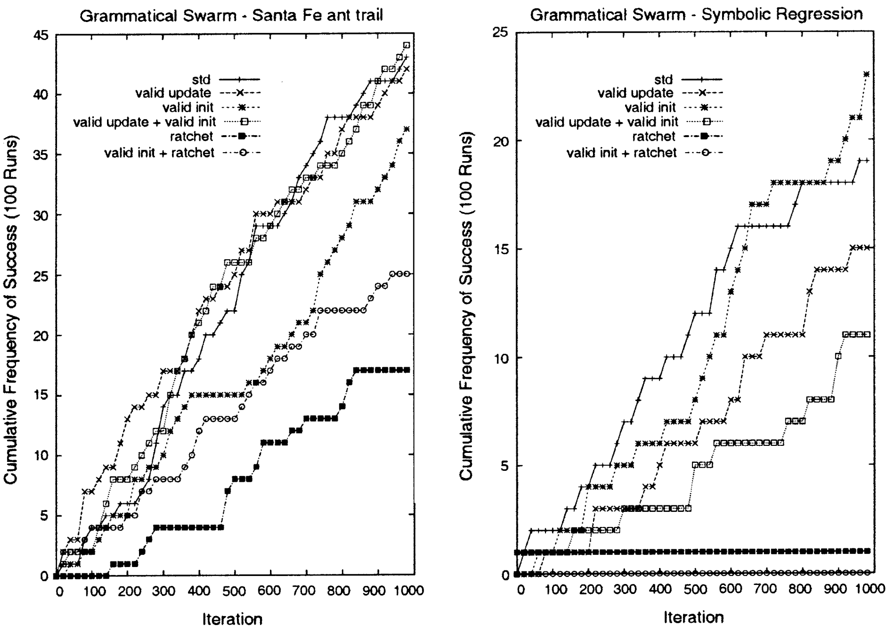


Figure 7. Plot of the cumulative frequency of success on the Santa Fe ant Trail (left) and on the quartic symbolic regression instance (right).

the ratchet strategies ("Ratchet" and "Valid Init + Ratchet") to the baseline GS approach, the baseline algorithm significantly outperformed the alternatives at the 95% confidence level on all the problems analyzed. The *t*-tests showed no differences in performance between the baseline GS algorithm to any of the other variants across all the problems analyzed (There was one exception to this where the "Valid Update + Valid Init" strategy was outperformed by the baseline GS algorithm on the Mastermind problem instance.).

In general, there is no clear winner amongst the remaining strategies over the standard Grammatical Swarm setup. This suggests that constraining movement of the particles where only fitness improvements are observed can have a negative impact on the performance of the GS algorithm. There is a slight improvement in performance in terms of the cumulative frequency of success for setups that adopted a valid initialization either on its own or in combination with a valid update step. This would point to future work which could investigate the use of a sensible initialization strategy as outlined in O'Neill and Ryan (2003) that would further improve the initialization of the population to ensure diversity in the structures of solutions generated.

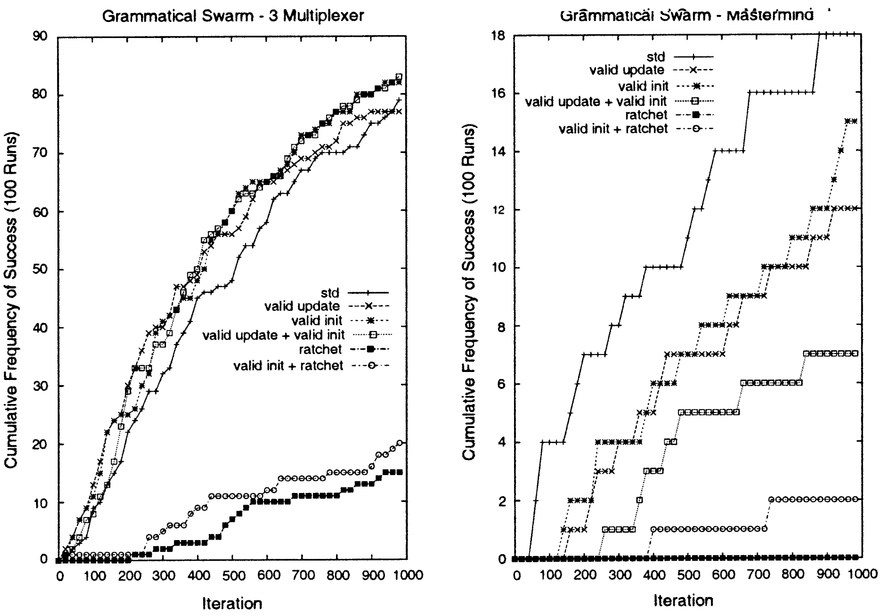


Figure 8. Plot of the cumulative frequency of success on the 3 multiplexer problem instance (left) and the mastermind problem using eight pins and four colors (right).

7. Conclusions and future work

This study demonstrates the feasibility of successfully generating computer programs using Grammatical Swarm through its application to a diverse set of benchmark program-generation problems. It thus represents the first instance of a Particle Swarm Algorithm being successfully adopted for the generation of programs. A performance comparison to Grammatical Evolution has shown that Grammatical Swarm is on a par with Grammatical Evolution, and is capable of generating solutions with much smaller populations, with a fixed-length vector representation, an absence of any crossover, and no concept of selection or replacement. Given the success of Grammatical Swarm at generating programs using a Particle Swarm algorithm, the relative importance of the social search metaphor warrants further investigation in Genetic Programming.

When analyzing the results presented, it is useful to remember that the Grammatical Evolution representation is variable-length, with individual's lengths restricted only by a computer's physical storage limitations. In the current implementation of Grammatical Swarm, fixed-length vectors are adopted from which a variable number of

Table 2. A comparison of the results obtained for Grammatical Swarm and the various Update and Initialization strategies across all the problems analyzed averaged over 100 runs

	Mean Best Fitness	SD	Median	Runs Successful
Santa Fe ant				
GS	75.24	16.64	88.0	43
Valid Update	74.49	17.65	88.0	42
Valid Init.	73.18	17.06	85.0	37
Valid Update + Valid Init.	75.6	16.09	88.0	44
Ratchet	56.36	22.71	46.0	17
Valid Init + Ratchet	58.48	22.48	48.50	25
Multiplexer				
GS	0.97	0.05	1.00	79
Valid Update	0.97	0.05	1.00	78
Valid Init.	0.98	0.05	1.00	83
Valid Update + Valid Init.	0.98	0.05	1.00	85
Ratchet	0.89	0.05	0.88	15
Valid Init + Ratchet	0.90	0.05	0.88	21
Symbolic regression				
GS	0.31	0.35	0.16	20
Valid Update	0.28	0.31	0.16	15
Valid Init.	0.34	0.36	0.17	23
Valid Update + Valid Init.	0.25	0.30	0.16	13
Ratchet	0.08	0.10	0.10	1
Valid Init + Ratchet	0.07	0.02	0.10	0
Mastermind				
GS	0.91	0.04	0.89	18
Valid Update	0.90	0.04	0.89	12
Valid Init.	0.91	0.04	0.89	15
Valid Update + Valid Init.	0.90	0.03	0.89	7
Ratchet	0.90	0.00	0.89	0
Valid Init + Ratchet	0.90	0.02	0.89	2

A *t*-test and bootstrap (re-sampling) *t*-test were performed at the 95% confidence level on the best fitness values of each strategy versus the baseline GS algorithm, with the result that the baseline GS algorithm outperformed both of the Ratchet strategies ("Ratchet" and "Valid Init + Ratchet") at the 95% confidence level on all the problems analyzed. None of the variants were found to be superior to the baseline GS algorithm.

dimensions can be generated. However, vectors have a hard-length constraint of 100 dimensions. It is common practice in a Genetic Programming approach such as Grammatical Evolution to allow the search algorithm to determine both the size and content of the generated programs, as we generally do not know *a priori* what size the solution program needs to be to solve the problem. We intend to implement a variable-length version of Grammatical Swarm that will allow the number of dimensions of a particle to increase and decrease over simulation time to overcome this current limitation.

Analysis of the behavior of Grammatical Swarm was analyzed in terms of the update of particles and initialization of the first population. It was determined that there may be scope to investigate sensible initialization strategies as there was a slight improvement in the success rate of runs when all particles were forced to correspond to a valid phenotype. Restricting the movement of particles to regions of the search space that only corresponded to better phenotypes resulted in a notable decrease in performance across all the problems analyzed.

The results presented are very encouraging for future development of the relatively simple Grammatical Swarm algorithm, and other potential Social or Swarm Programming variants.

References

Banzhaf W, Nordin P, Keller RE and Francone FD (1998) Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, San Mateo, California

Bonabeau E, Dorigo M and Theraulaz G (1999) Swarm Intelligence: From Natural to Artificial Systems. Oxford University Press, Oxford

Brabazon A and O'Neill M (2006) Biologically Inspired Algorithms for Financial Modelling. Springer, Berlin

Brabazon A, Silva A, de Sousa TF, ONeill M, Matthews R and Costa E (2005) Investigating strategic inertia using OrgSwarm. Informatica 29: 125–141

Kennedy J, Eberhart R and Shi Y (2001) Swarm Intelligence. Morgan Kauff-man, San Mateo, California

Kennedy J and Eberhart R (1995) Particle swarm optimization, Proceedings of the IEEE International Conference on Neural Networks, December 1995, pp. 1942–1948, IEEE Press

Koza JR (1992) Genetic Programming. MIT Press

Koza JR (1994) Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press

Koza JR, Andre D, Bennett FH III and Keane M (1999) Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufmann, San Mateo, California

Koza JR, Keane M, Streeter MJ, Mydlowec W, Yu J, Lanza G (2003) Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers

Langdon WB and Poli R (1998) Why ants are hard. In: Genetic Programming 1998: Proceedings of the Third Annual Conference. University of Wisconsin, Madison, Wisconsin, USA, pp. 193–201, Morgan Kaufmann, San Mateo, California

O'Neill M and Brabazon A (2004) Grammatical Swarm. In: LNCS 3102 Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2004. Seattle, WA, USA, pp. 163–174, Springer, Berlin

O'Neill M and Ryan C (2003) Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers

O'Neill M (2001) Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution. PhD thesis, University of Limerick

O'Neill M and Ryan C (2001) Grammatical evolution. IEEE Transactions on Evolutionary Computation 5(4): 349–358

O'Neill M, Ryan C, Keijzer M and Cattolico M (2003) Crossover in grammatical evolution. Genetic Programming and Evolvable Machines 4(1): 67–93

Ryan C, Collins JJ, O'Neill M (1998) Grammatical evolution: evolving programs for an arbitrary language. Proceedings of the First European Workshop on GP. pp. 83–95, Springer-Verlag, Berlin

Silva A, Neves A, Costa E (2002) An empirical comparison of particle swarm and predator prey optimisation. In: LNAI 2464, Artificial Intelligence and Cognitive Science, the 13th Irish Conference AICS 2002. pp. 103–110, Limerick, Ireland, Springer, Berlin