Grammatical Swarm

Michael O'Neill[1] and Anthony Brabazon[2]

[1] Biocomputing and Developmental Systems Group
University of Limerick, Ireland
`Michael.ONeill@ul.ie`
[2] University College Dublin, Ireland
`Anthony.Brabazon@ucd.ie`

Abstract. This proof of concept study examines the possibility of specifying the construction of programs using a Particle Swarm algorithm, and represents a new form of automatic programming based on Social Learning, *Social Programming* or *Swarm Programming*. Each individual particle represents choices of program construction rules, where these rules are specified using a Backus-Naur Form grammar. The results demonstrate that it is possible to generate programs using the Grammatical Swarm technique.

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 [1]. 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 [2,3]. 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. In the Grammatical Swarm (GS) approach, each particle or real-valued vector, represents choices of program construction rules specified as production rules of a Backus-Naur Form grammar.
This approach is grounded in the linear program representation adopted in Grammatical Evolution (GE) [4,5,6,7,8], which uses grammars to guide the construction of syntactically correct programs, specified by variable-length genotypic binary or integer strings. The search heuristic adopted with GE is thus a variable-length Genetic Algorithm. In the Grammatical Swarm 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 could dub *Social Programming*, or *Swarm Programming*. It is interesting to note that this approach is completely devoid of any crossover operator characteristic of Genetic Programming.
The remainder of the paper is structured as follows. Before describing the mechanism of Grammatical Swarm 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 provides some discussion of the results, and finally section 7 details conclusions and 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.' [2](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 (*pbest*), and knows the location of 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) (*gbest*). 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 pbest and gbest. 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 PSA exist. The following paragraphs provide a description of the basic *continuous* version described by [2].

i. Initialize 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 *pbest*.
iii. Determine the location of the particle with the highest fitness and revise *gbest* if necessary.
iv. For each particle, calculate its velocity according to equation 1.
v. Update the location of each particle.
vi. Repeat steps ii - v until stopping criteria are met.

The update algorithm for the velocity, v, of each dimension, i, of a vector is:

$$v_i^{`} = (w * v_i) + (c1 * R_1 * (p_{best} - p_i)) + (c2 * R_2 * (g_{best} - p_i)) \quad (1)$$

where,

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

$c1 = 1.0$ is the weight associated with the personal best dimension value, $c2 = 1.0$ the weight associated with the global best dimension value, R_1 and R_2 are a random real number between 0 and 1, p_{best} is the vector's best dimension value to date, p_i is the vector's current dimension value, g_{best} is the best dimension value globally, $wmax = 0.9$, $wmin = 0.4$, $itermax$ is the total number of iterations in the simulation, $iter$ is the current iteration value, and $vmax$ places bounds on the magnitude of the updated velocity value.
Once the velocity update for particle i is determined, its position is updated and pbest is updated if necessary.

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

After all particles have been updated, a check is made to determine whether gbest needs to be updated.

$$\hat{y} \in (y_0, y_1, ..., y_n) | f(\hat{y}) = \max\ (f(y_0), f(y_1), ..., f(y_n)) \tag{4}$$

3 Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary algorithm that can evolve computer programs in any language [4,5,6,7,8], and can be considered a form of grammar-based genetic programming. Rather than representing the programs as parse trees, as in GP [9,10,11,12,13], a linear genome representation is used. 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. 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 or sentence(s)), according to the production rules. A simple example BNF grammar is given below, where `<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:

$$Rule = Codon\ Value\ \%\ Num.\ Rules$$

where % represents the modulus operator.

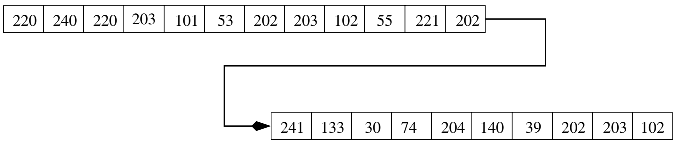


Fig. 1. An example GE individuals' genome represented as integers for ease of reading.

Given the example individuals' genome (where each 8-bit codon is represented as an integer for ease of reading) in Fig.1, 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 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. Fig.2 draws an analogy between GE's mapping process and the molecular biological processes of transcription and translation. A full description of GE can be found in [4].

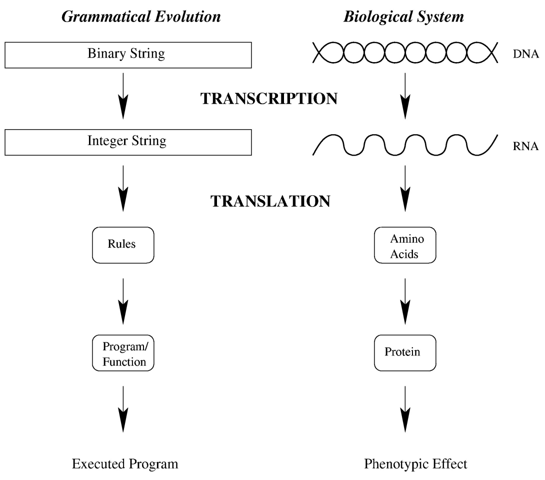


Fig. 2. A comparison between Grammatical Evolution and the molecular biological processes of transcription and translation. The binary string of GE is analogous to the double helix of DNA, each guiding the formation of the phenotype. In the case of GE, this occurs via the application of production rules to generate the terminals of the compilable program. In the biological case by directing the formation of the phenotypic protein by determining the order and type of protein subcomponents (amino acids) that are joined together.

4 Grammatical Swarm

Grammatical Swarm (GS) adopts a Particle Swarm learning algorithm coupled to a Grammatical Evolution (GE) genotype-phenotype mapping to generate programs in an arbitrary language. The update equations for swarm algorithm are as described earlier, with additional constraints placed on the velocity and dimension values, such that velocities are bound to VMAX=$\pm$255, and each dimension is bound to the range 0 to 255. 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 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.

5 Experiments and Results

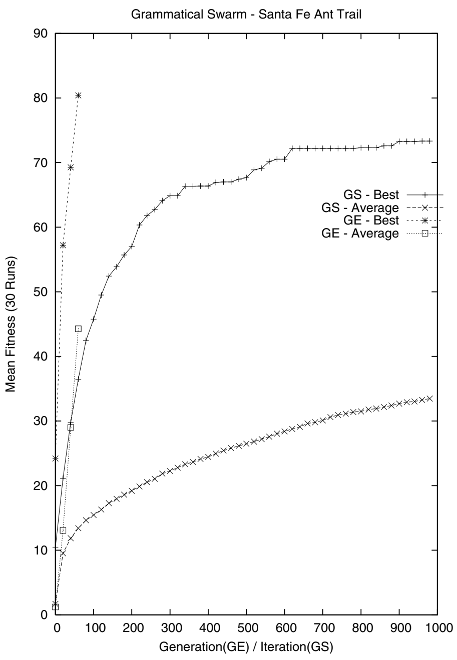
A diverse selection of benchmark programs from the literature on evolutionary automatic programming are tackled using Grammatical Swarm to demonstrate proof of concept for the GS methodology. The parameters adopted across the following experiments are c1 = 1.0, c2 = 1.0, wmax = 0.9, wmin = 0.4, CMIN = 0 (minimum value a coordinate may take), CMAX = 255 (maximum value a coordinate may take), and VMAX = CMAX (i.e., velocities are bound to the range +VMAX to -VMAX). In addition, a swarm size of 30 running for 1000 iterations is used, where each particle is represented by a vector with 100 elements.
The same problems are also tackled with Grammatical Evolution in order to get some indication of how well Grammatical Swarm is performing at program generation in relation to the more traditional variable-length Genetic Algorithm-driven 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, and using typical population sizes from the literature adopted for each method. 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 [14]. 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 problem is different to the ones used later for symbolic regression and the multiplexer problem 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 30 runs can be seen in Fig.3. As can be seen, convergence towards the best fitness occurs, and a number of runs successfully obtain the correct solution.



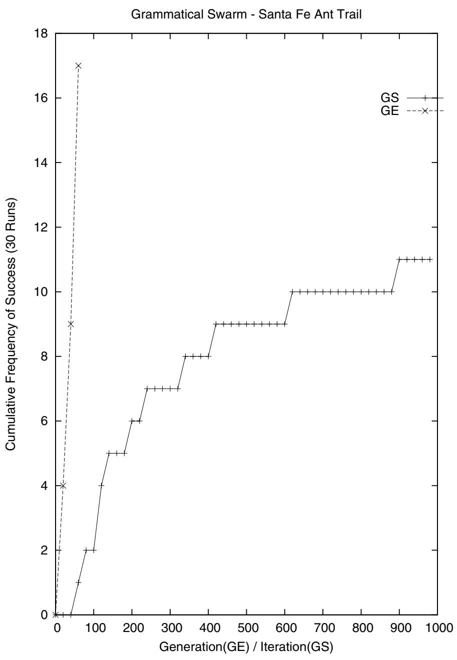


Fig. 3. Plot of the mean fitness on the Santa Fe Ant Trail problem instance (left), and the cumulative frequency of success (right).

5.2 Quartic Symbolic Regression

The target function is $f(a) = a + a^2 + a^3 + a^4$, 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 30 runs can be seen in Fig.4. As can be seen, a number of runs successfully find the correct solution to the problem, with convergence towards the best fitness occurring on average.

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

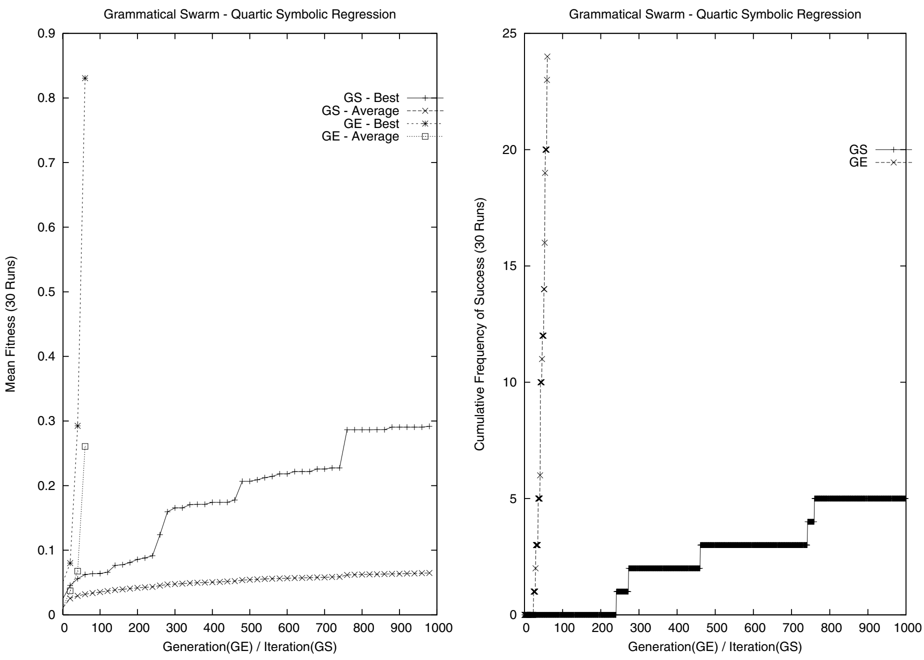


Fig. 4. Plot of the mean fitness on the quartic symbolic regression problem instance (left), and the cumulative frequency of success (right).

problem is to discover a boolean expression that behaves as a 3 Multiplexer. There are 8 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
```

A plot of the mean best fitness over 30 runs can be seen in Fig.5. As can be seen, convergence towards the best fitness occurs, and a number of runs successfully evolve correct solutions.

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

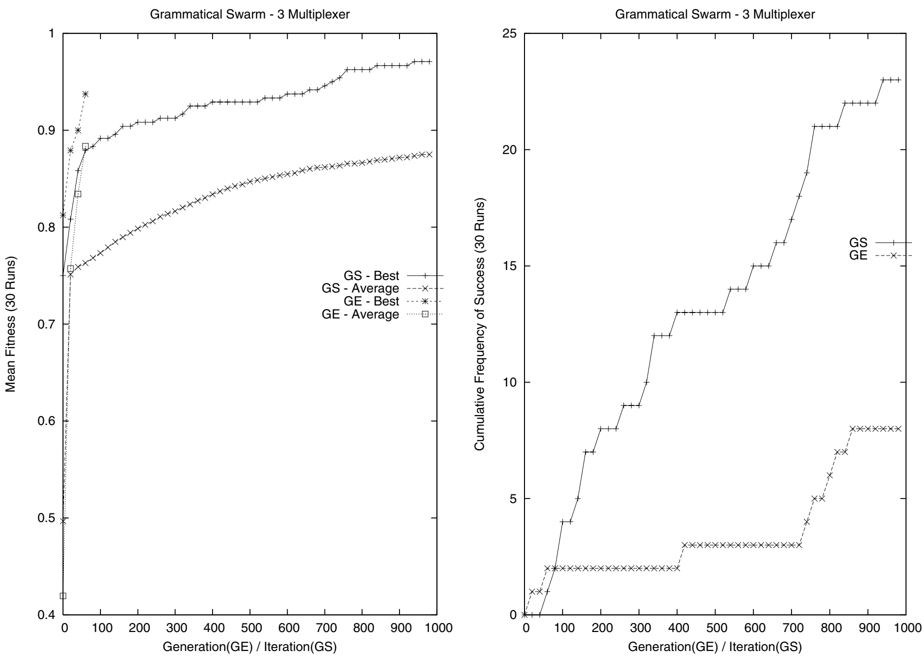


Fig. 5. Plot of the mean fitness on the 3 multiplexer problem instance (left), and the cumulative frequency of success (right).

has the correct color, regardless of its position. If all pins are in the correct order than 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 color, but in the wrong positions. The difficulty of this problem is controlled by the number of pins and the number of colors in the target combination. The instance tackled here uses 4 colors and 8 pins with the following values 3 2 1 3 1 3 2 0.
Results are provided in Fig. 6 and the grammar adopted is as follows.

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

6 Discussion

Table 1 provides a summary and comparison of the performance of Grammatical Swarm and Grammatical Evolution on each of the problem domains tackled. In two out of the four problems Grammatical Evolution outperforms Grammatical Swarm, Grammatical Swarm outperforms Grammatical Evolution on one problem instance, and there is a tie between the methods on the Mastermind problem. The key finding is that the results demonstrate proof of concept that Grammatical Swarm can successfully generate solutions to problems of interest.

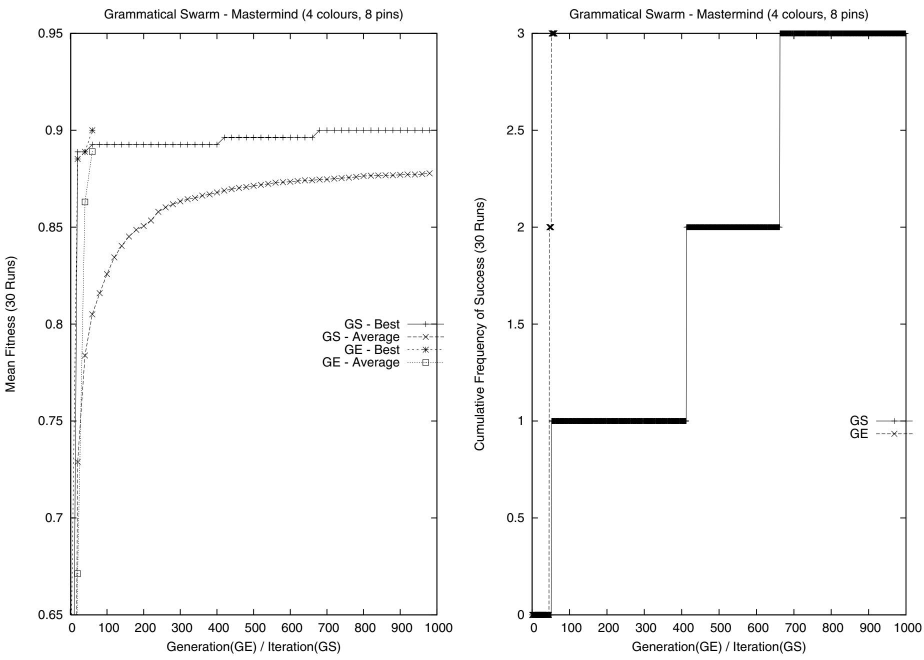


Fig. 6. Plot of the mean best and mean average fitness (left) and the cumulative frequency of success (right) on the Mastermind problem instance using 8 pins and 4 colors. Fitness is defined as the points score of a solution divided by the maximum possible points score.

In this initial study, we have not attempted parameter optimization 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 [15], and we suspect that a significant improvement in Grammatical Swarms' 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.
When analyzing the results presented one has to consider the fact that the Grammatical Evolution representation is variable-length with individuals' lengths restricted only by the machines physical storage limitations. In the current implementation of Grammatical Swarm fixed-length vectors are adopted in which a variable number of dimensions can be used, however, vectors have a hard length constraint of 100 elements. 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.

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

	Mean Best Fitness (Std.Dev.)	**Mean Average Fitness (Std.Dev.)**	**Successful Runs**
Santa Fe ant			
GS	73.3 (17.6)	33.6 (3.32)	11
GE	**80.4** (14.4)	**44.3** (5.7)	**17**
Multiplexer			
GS	**0.97** (0.05)	0.88 (0.01)	**23**
GE	0.94 (0.06)	0.88 (0.02)	15
Symbolic Regression			
GS	0.29 (0.35)	0.07 (0.02)	5
GE	**0.83** (0.33)	**0.26** (0.26)	**24**
Mastermind			
GS	**0.9** (0.03)	**0.88** (0.013)	**3**
GE	**0.9** (0.03)	**0.89** (0.001)	**3**

7 Conclusions and Future Work

This study demonstrates the feasibility of the generation of computer programs using Grammatical Swarm over four different problem domains. As such a new form of automatic programming based on social learning is introduced, which could be termed *Social Programming*, or *Swarm Programming*. While a performance comparison to Grammatical Evolution has shown that Grammatical Swarm is outperformed on two of the problems analyzed, the ability of Grammatical Swarm to generate solutions with such small populations, with a fixed-length vector representation, an absence of any crossover, no concept of selection or replacement, and without optimization of the algorithm's parameters is very encouraging for future development of the much simpler Grammatical Swarm, and other potential Social or Swarm Programming variants. Future work will involve developing a variable-length Particle Swarm algorithm to remove Grammatical Swarms length constraint, conducting an investigation into swarm diversity, the impact of a continuous encoding over a discrete encoding variant such as presented in [16], and considering the implications of a social learning approach to the automatic generation of programs.

References

1. Bonabeau, E., Dorigo, M. and Theraulaz, G. (1999). *Swarm Intelligence: From natural to artificial systems*, Oxford: Oxford University Press.
2. Kennedy, J., Eberhart, R. and Shi, Y. (2001). *Swarm Intelligence*, San Mateo, California: Morgan Kauffman.
3. Kennedy, J. and Eberhart, R. (1995). Particle swarm optimization, *Proc. of the IEEE International Conference on Neural Networks*, pp.1942-1948.
4. O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers.

5. O'Neill, M. (2001). *Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution.* PhD thesis, University of Limerick, 2001.
6. O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation.* Vol. 5, No.4, 2001.
7. O'Neill, M., Ryan, C., Keijzer M., Cattolico M. (2003). Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*, Vol. 4 No. 1. Kluwer Academic Publishers, 2003.
8. Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proc. of the First European Workshop on GP*, 83-95, Springer-Verlag.
9. Koza, J.R. (1992). *Genetic Programming.* MIT Press.
10. Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.
11. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann.
12. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufmann.
13. Koza, J.R., Keane, M., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers.
14. Langdon, W.B., and Poli, R. (1998). Why Ants are Hard. In *Genetic Programming 1998: Proc. of the Third Annual Conference*, University of Wisconsin, Madison, Wisconsin, USA, pp. 193-201, Morgan Kaufmann.
15. 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.
16. Kennedy, J., and Eberhart, R. (1997). A discrete binary version of the particle swarm algorithm. *Proc. of the 1997 Conference on Systems, Man, and Cybernetics*, pp. 4104-4109. Piscataway, NJ: IEEE Service Center.