Grammatical Swarm: A variable-length Particle Swarm Algorithm

Michael O'Neill[1], Finbar Leahy[2], and Anthony Brabazon[1]

[1] University College Dublin, Belfield, Dublin 4, Ireland.
m.oneill@ucd.ie, anthony.brabazon@ucd.ie
[2] University of Limerick, Limerick, Ireland.
finbarleahy@gmail.com

This study examines a variable-length Particle Swarm Algorithm for Social Programming. The Grammatical Swarm algorithm is a form of Social Programming as it uses Particle Swarm Optimisation, a social swarm algorithm, for the automatic generation of programs. This study extends earlier work on a fixed-length incarnation of Grammatical Swarm, where each individual particle represents choices of program construction rules, where these rules are specified using a Backus-Naur Form grammar. A selection of benchmark problems from the field of Genetic Programming are tackled and performance is compared to that of fixed-length Grammatical Swarm and of Grammatical Evolution. The results demonstrate that it is possible to successfully generate programs using a variable-length Particle Swarm Algorithm, however, based on the problems analysed it is recommended that the simpler bounded Grammatical Swarm be adopted.

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 [2]. 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 [6, 7]. 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 a variable-length Particle Swarm Algorithm for the automated construction of a program using a Social Programming model. The performance of this variable-length Particle Swarm approach is compared to its fixed-length counterpart [15, 17] and to Grammatical Evolution on a number of benchmark problems. In the Grammatical Swarm (GS) methodology developed in this paper, 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 Genetic Programming representation adopted in Grammatical Evolution (GE) [18], 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 a variable-length Genetic Algorithm. A variable-length representation is adopted as the size of the program is not known a-priori and must itself be determined automatically. 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 characteristic of Genetic Programming.

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, and finally section 6 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.' [6](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.

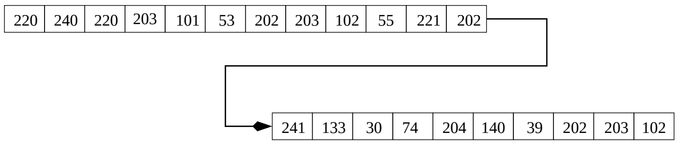

Fig. 1. An example GE individuals' genome represented as integers for ease of reading.

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.
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:

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

where

$$w = wmax - ((wmax - wmin)/itermax) * iter \quad (2)$$

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). The term 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 & 5).

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

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

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

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

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

3 Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary algorithm that can evolve computer programs in any language [18, 19, 20, 21, 22], and can be considered a form of grammar-based genetic programming. GE has enjoyed particular success in the domain of Financial Modelling [3] amongst numerous other applications including Bioinformatics, Systems Biology, Combinatorial Optimisation and Design [16, 13, 5, 4]. Rather than representing the programs as parse trees, as in GP [8, 9, 1, 10, 11], 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, and can 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:

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

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

Given the example individual's 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. This codon is then used as follows: $240 \,\%\, 2 = 0$ 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 [18], and some more recent developments are covered in [3, 14].

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 [15]. 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 contrast to earlier studies on GS this study adopts variable-length vectors. 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. Although the vectors were bounded in length in earlier studies not all elements were necessarily used to construct a program during the mapping process, and as such the programs generated were variable in size.

4.1 Variable-length Particle Strategies

Four different approaches to a variable-length particle swarm algorithm were investigated in this study.

Strategy I

Each particle in the swarm is compared to the global best particle (gbest) to determine if there is a difference between the length of the particle's vector and the length of the gbest vector. If there is no difference between the vector sizes then a length update is not required and the algorithm simply moves on and compares the next particle to gbest. However, when there is a difference between the vector lengths, the particle is either extended or truncated. If the current particles, p_i vector length is shorter than the length of gbest, elements are added to the particle's vector extending it so that it is now equivalent in length to that of gbest. The particle's new elements contain values which are copied directly from gbest. For example, if gbest is a vector containing fifty elements and the current particle has been extended from forty five to fifty elements then the values contained in the last 5 elements (46-50) of gbest are copied into the five new elements of the current particle. If the particle has a greater number of elements than the gbest particle, then the extra elements are simply truncated so that both gbest and the current particle have equivalent vector lengths.

Strategy II

This strategy is similar to the first strategy, the only difference is the method in which the new elements are copied. In the first strategy, when the current particle, p_i is extended the particle's new elements are populated by values which are copied directly from gbest. In Strategy II, values are not copied from gbest instead random numbers are generated in the range [cmin, cmax] and these values are copied into each of p_i's new elements.

Strategy III

The third strategy involves the use of probabilities. Given a specified probability, the length of the particle is either increased or decreased. A maximum of one elements can only be changed at a time i.e. either an element is added or removed from the current particle, p_i. If p_i is longer than gbest then the last element of p_i is discarded. If p_i is shorter than gbest then p_i is increased by adding an extra element to its vector. In this situation the new element takes the value of a random number in the range [cmin, cmax].

Strategy IV

The fourth strategy involves the generation of a random number to determine the number of elements that will be added to or removed from the current particle, p_i. If the length of p_i is shorter than the length of gbest the difference, dif between the length of gbest and the length of p_i is calculated. Then a random integer is generated in the range [0, dif]. The result of this calculation is then used to determine how many elements will be truncated from p_i. A similar strategy is applied when the length of p_i is smaller than the length of gbest. However, in this case the random number generated is used to determine the number of elements that p_i will be extended by. After p_i is extended, each of these extended elements are then populated with random numbers generated in the range [cmin, cmax].

A strategy is not applied every time it was possible to modify the current particle (p_i), instead applying a strategy is determined by the outcome of a certain probability function i.e. the outcome of this function is used to determine if a strategy is to be applied to p_i. In our current implementation, a probability of 0.5 was selected. Therefore 50% of the time a length-modifying strategy is applied and 50% of the time the length of p_i is not modified.

For each particle in the swarm, a random number in the range [1,100] is generated, which determines its initial length in terms of the number of codons.

5 Proof of Concept Experiments & Results

A diverse selection of benchmark programs from the literature on Genetic Programming are tackled using Grammatical Swarm to demonstrate proof of concept for the variable-length 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 is used, and 100 independent runs are performed for each experimental setup with average results being reported.

The same problems are also tackled with GS's fixed-length counterpart (using 100 dimensions) and GE in order to determine how well the variable-length GS algorithm 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.

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 [12]. 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();
```

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
```

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 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
```

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 target combination. The instance tackled here uses 4 colors and 8 pins with the following target values 3 2 1 3 1 3 2 0.

The grammar adopted is as follows.

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

5.5 Results

Results averaged over 100 runs showing the best fitness, and the cumulative frequency of success for the four variable length grammatical swarm (VGS) variants are presented in Figures 2, 3, 4, and 5.

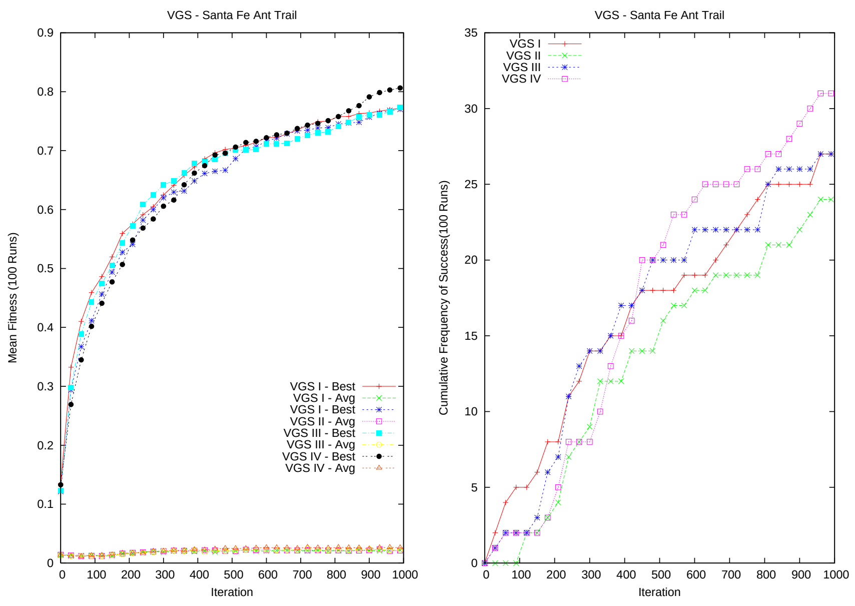

Fig. 2. Plot of the mean fitness on the Santa Fe Ant trail problem instance (left), and the cumulative frequency of success (right).

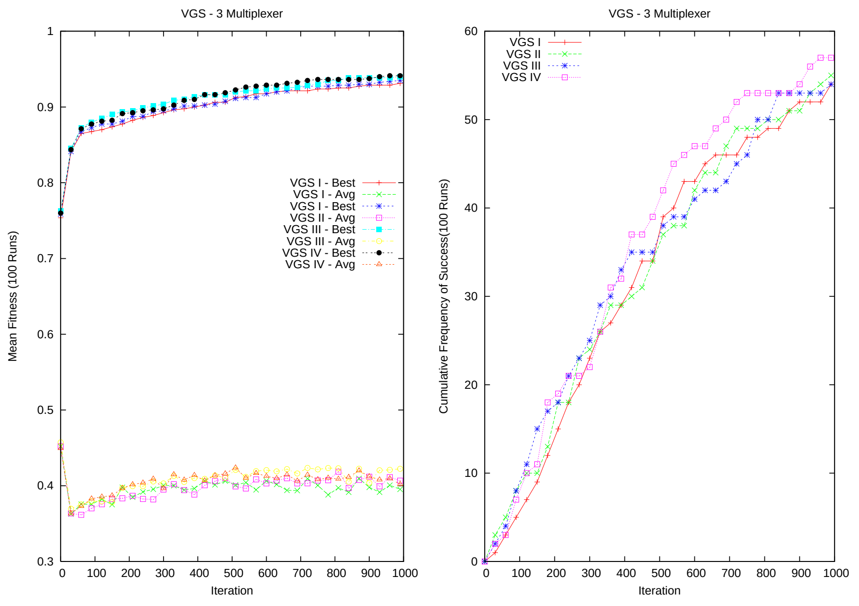

Fig. 3. Plot of the mean fitness on the 3 multiplexer problem instance (left), and the cumulative frequency of success (right).

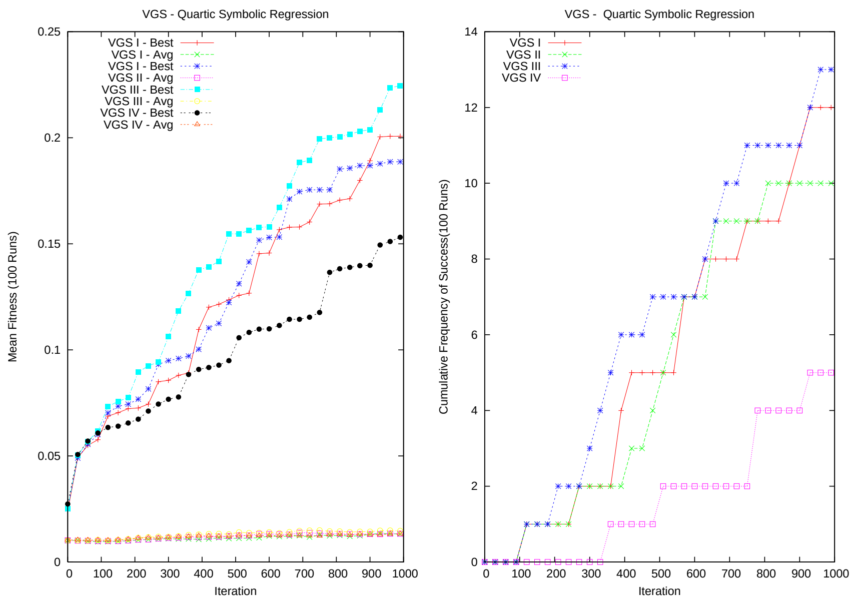

Fig. 4. Plot of the mean fitness on the Quartic Symbolic Regression problem instance (left), and the cumulative frequency of success (right).

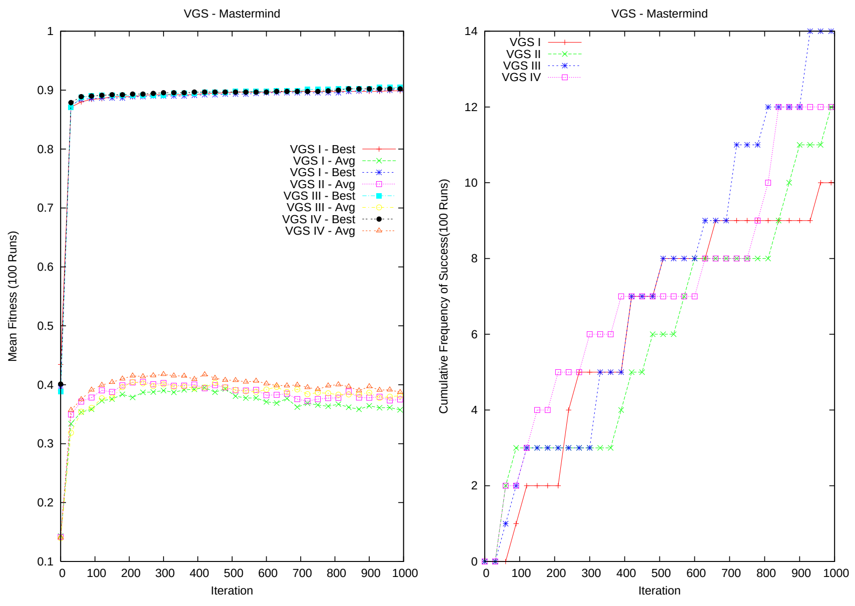

Fig. 5. Plot of the mean fitness on the Mastermind problem instance (left), and the cumulative frequency of success (right).

Tables 1, 2, 3 and 4 outline a comparison of the results across the four variable-length particle swarm strategies analysed in this study. While there is no clear winner across all four problems strategies, III and IV were the most successful overall, with strategy IV producing best performance on the Santa Fe ant and Multiplexer problems, while strategy III had the better performance on the Symbolic Regression and Mastermind instances. It is interesting to note that the mean length of the gbest particle never grows beyond 65 codons at the last iteration across all four problems, demonstrating that bloat does not appear to have impacted on these results.

Table 1. A comparison of the results obtained for the four different variable-length Particle Swarm Algorithm strategies on the Santa Fe Ant trail.

	Mean Best Fitness	**Successful Runs**	**Mean gbest Codon Length**
Strategy			
I	.77	27	50
II	.76	24	51
III	.78	27	51
IV	.8	31	61

Table 2. A comparison of the results obtained for the four different variable-length Particle Swarm Algorithm strategies on the Multiplexer problem instance.

	Mean Best Fitness	**Successful Runs**	**Mean gbest Codon Length**
Strategy			
I	.93	54	49
II	.94	55	52
III	.94	54	57
IV	.94	57	53

Table 3. A comparison of the results obtained for the four different variable-length Particle Swarm Algorithm strategies on the quartic symbolic regression problem instance.

	Mean Best Fitness	**Successful Runs**	**Mean gbest Codon Length**
Strategy			
I	.2	12	45
II	.19	10	49
III	.23	13	55
IV	.15	5	54

Table 4. A comparison of the results obtained for the four different variable-length Particle Swarm Algorithm strategies on the Mastermind problem.

	Mean Best Fitness	**Successful Runs**	**Mean gbest Codon Length**
Strategy			
I	.89	10	61
II	.9	12	57
III	.9	14	65
IV	.9	12	60

5.6 Summary

Table 5 provides a summary and comparison of the performance of the fixed and variable-length forms of GS and GE on each of the problem domains tackled. The best variable-length strategy outperforms GE on the Mastermind instance and has a similar performance to the fixed-length form of GS. On the other three problems the fixed-length form of GS outperforms variable-length GS in terms of the number of successful runs finding the target solution. On both the Santa Fe ant and Symbolic Regression problems, GE outperforms GS. The key finding is that the results demonstrate proof of concept that a variable-length particle swarm algorithm can successfully generate solutions to problems of interest. In this initial study, we have not attempted parameter optimization for the various variable-length strategies and this may lead to further improvements of the variable-length particle swarm algorithm. 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. Based on the findings in this study there is no clear winner between the bounded and variable-length forms of GS, and as such the recommendation at present would be to adopt the simpler bounded GS, although future investigations may find in the variable-length algorithm's favour.

Table 5. A comparison of the results obtained for Grammatical Swarm and Grammatical Evolution across all the problems analyzed.

	Successful Runs
Santa Fe ant	
GS (variable)	31
GS (bounded)	38
GE	58
Multiplexer	
GS (variable)	57
GS (bounded)	87
GE	56
Symbolic Regression	
GS (variable)	13
GS (bounded)	28
GE	85
Mastermind	
GS (variable)	14
GS (bounded)	13
GE	10

6 Conclusions & Future Work

This study demonstrates the feasibility of successfully generating computer programs using a variable-length form of Grammatical Swarm, and demonstrates its application to a diverse set of benchmark program-generation problems. 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. A performance comparison of the variable-length and fixed-length forms of Grammatical Swarm reveal that the simpler fixed-length version is superior for the experimental setups and problems examined here.

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

1. Banzhaf, W., Nordin, P., Keller, R.E. and Francone, F.D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann.
2. Bonabeau, E., Dorigo, M. and Theraulaz, G. (1999). *Swarm Intelligence: From natural to artificial systems*, Oxford: Oxford University Press.
3. Brabazon, A. and O'Neill, M. 2006. *Biologically Inspired Algorithms for Financial Modelling.* Springer.
4. Cleary, R. and O'Neill, M. 2005. An Attribute Grammar Decoder for the 01 MultiConstrained Knapsack Problem. In LNCS 3448 *Pr oc. of Evolutionary Computation in Combinatorial Optimization EvoCOP 2005*, pp.34-45, Lausanne, Switzerland. Springer.
5. Hemberg, M. and O'Reilly, U-M. 2002. GENR8 - Using Grammatical Evolution In A Surface Design Tool. In *Proc. of the First Gra mmatical Evolution Workshop GEWS2002*, pp.120-123. New York City, New York, US. ISGEC.
6. Kennedy, J., Eberhart, R. and Shi, Y. (2001). *Swarm Intelligence*, San Mateo, California: Morgan Kauffman.
7. Kennedy, J. and Eberhart, R. (1995). Particle swarm optimization, *Proceedings of the IEEE International Conference on Neural Networks*, December 1995, pp.1942-1948.
8. Koza, J.R. (1992). *Genetic Programming.* MIT Press.
9. Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.
10. Koza, J.R., Andre, D., Bennett III and F.H., Keane, M. (1999). *Genetic Programming 3: Darwinian Invention and Problem So lving.* Morgan Kaufmann.
11. Koza, J.R., Keane, M., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G. (2003). *Genetic Programming IV: Routine Human-Co mpetitive Machine Intelligence.* Kluwer Academic Publishers.
12. Langdon, W.B. and Poli, R. (1998). Why Ants are Hard. In *Genetic Programming 1998: Proceedings of the Th ird Annual Conference*, University of Wisconsin, Madison, Wisconsin, USA, pp. 193-201, Morgan Kaufmann.
13. Moore, J.H. and Hahn, L.W. (2004). Systems Biology Modeling in Human Genetics Using Petri Nets and Grammatical Evolution . In LNCS 3102 *Proc. of the Genetic and Evolutionary Computation Conference GECCO 2004*, Seattle, WA, USA, pp.392-401. Springer.
14. O'Neill, M. and Brabazon, A. (2005). Recent Adventures in Grammatical Evolution. In *Computer Methods and Systems CMS'05*, Krakow, Poland, pp.245-252. Oprogramowanie Naukowo-Techniczne.
15. O'Neill, M. and Brabazon, A. (2004). Grammatical Swarm. In LNCS 3102 *Proc. of the Genetic and Evolutionary Computation Conferen ce GECCO 2004*, Seattle, WA, USA, pp.163-174. Springer.

16. O'Neill, M., Adley, C. and Brabazon, A. (2005). A Grammatical Evolution Approach to Eukaryotic Promoter Recognition. In *Proc. of Bioinformatics INFORM 2005*, Dublin City University, Dublin, Ireland.
17. O'Neill, M., Brabazon, A. and Adley, C. (2004). The automatic generation of programs for Classification using Grammatical Swarm. In *Proc. of the Congress on Evolutionary Computation CEC 2004*, Portland, OR, USA, pp.104-110. IEEE.
18. O'Neill, M. and Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers.
19. O'Neill, M. (2001). *Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution.* PhD thesis, University of Limerick, 2001.
20. O'Neill, M. and Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation.* 2001.
21. O'Neill, M., Ryan, C., Keijzer M. and Cattolico M. (2003). Crossover in Grammatical Evolution. *Genetic Programming and E volvable Machines*, Vol. 4 No. 1. Kluwer Academic Publishers, 2003.
22. Ryan, C., Collins, J.J. and O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proc. of the First European Workshop on GP*, 83-95, Springer-Verlag.
23. Silva, A., Neves, A. and Costa, E. (2002). An Empirical Comparison of Particle Swarm and Predator Prey Optimisation. In *LN AI 2464, Artificial Intelligence and Cognitive Science, the 13th Irish Conference AICS 2002*, pp. 103-110, Limerick, Ireland, Springer.