Improving Module Identification and Use in Grammatical Evolution

Aidan Murphy
Biocomputing and Developmental Systems Group,
CSIS Department,
University of Limerick, Ireland
aidan.murphy@ul.ie

Conor Ryan
Biocomputing and Developmental Systems Group,
CSIS Department,
University of Limerick, Ireland
conor.ryan@ul.ie

***Abstract*—Exploiting patterns within a solution or reusing certain functionality is often necessary to solve certain problems. This paper proposes a new method for identifying useful modules. Modules are only considered if they are prevalent in the population and they are seen to have a positive effect on an individual's fitness. This is achieved by finding the covariance of an individual's fitness with the presence of a particular subtree in the overall expression.**

While there are many successful systems that dynamically add modules during Genetic Programming (GP) runs, doing so is not trivial for Grammatical Evolution (GE), due to the fact that it employs a mapping process to produce individuals from binary strings, which makes it difficult to dynamically change the mapping process during a run.

We adopt a multi-run approach which only has a single stage of module addition to mitigate the problems associated with continuously adding newly found functionality to a grammar. Based on the well-known Price Equation, our system explores the covariance between traits to identify useful modules, which are added to the grammar, before the system is restarted. Grammar Augmentation through Module Encapsulation (GAME) was tested on seven problems from three different domains and was observed to significantly improve the performance on 3 problems and never showing harmful effects on any problem. GAME found the best individual in 6 of the 7 experiments.

***Index Terms*—Grammatical evolution, Modularity, Encapsulation**

I. Introduction

Many problems involve patterns, repeating structures or necessary components to be present in order to obtain their solutions. A human programmer will often write classes or functions and repeatedly use them to solve a problem. GP [1] is an evolutionary computation (EC) [2] technique for evolving computer programs and, therefore, a principal goal of GP has been to devise procedures to identify these structures or building blocks (typically in the form of sub-trees) and investigate how to best utilise them and protect them from the potentially harmful effects of the evolutionary procedures of crossover and mutation. Pinpointing and exploiting this modularity has been shown to greatly increase the effectiveness in finding solutions and has become a necessary feature to solve some problems in automatic programming [3].

However, the benefits of subtree discovery are not limited to finding optimum solutions. They have also been shown to reduce bloat in final programs, mitigate bias and as a means to transfer knowledge from one problem to another in the same domain. This would reduce the need for processes such as post-run intron removal [4]. More recently it has been put forward as a key ingredient for GP to enter the domain of model interpretability, a field of growing interest in academia, governance and business, and of which there has been very little written to date in GP or EC in general [5]. Decomposability, the capacity to fragment a large solution into smaller pieces, has been highlighted as a very desirable trait of any explainable model [6]. Thus, perhaps unique to this domain, identification of these subprograms or routines both helps in the search for the best program while also potentially boosting the interpretability of the solution generated.

Grammatical Evolution (GE) [7] is another very popular evolutionary computation search technique which uses a grammar, generally a context free grammar (CFG) written in Backus-Naur form (BNF), to find syntactically correct executable programs which solve a given problem. GE individuals are binary strings which are mapped onto derivation trees using the grammar, [8], so the search operators are performed on the strings, like standard genetic algorithms (GAs), and not the actual programs. This separation between the search space, at the genome level, and the program space, at the phenotype level, is seen as one of GE's many advantages over regular GP as it greatly simplifies the search operation and makes the addition of other search techniques, such as particle swarm [9], possible. One of the most desirable and important advantages GE has over other search techniques is its ability to produce programs for any arbitrary language.

Many approaches for identifying useful modularity rely on the evolution itself to find useful substructures [3]. The hypothesis being if a particular subtree exists in many solutions it must be useful, however, for many techniques the best selection method for finding subtrees has been random choice.

This paper introduces a novel approach to subtree identification. The selection procedure assigns a fitness to a subtree by finding its frequency in the population. If the covariance between the subtree being present in a full tree and the full tree's fitness is negative this fitness is changed to 0. Covariance takes into account both the fitness of the trees it is present in but also the fitness of the tree it does not appear in. This approach, uniquely, would also allow for the identification and

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

removal of harmful substructures in the population. In the case of GE, this could mean the complete removal of a terminal from the grammar.

The separation between the search and program space can mean the genetic operators are particularly destructive to expressions in GE. This phenomenon is known as *ripple crossover* [10] and creates a further imperative to encapsulate and save useful functionality once it is found. This creates difficulties when new terminals (in the shape of encapsulated sub-trees) are added to the grammar, specifically, the meaning of codons may change when an individual is remapped. The previous attempt at GE encapsulation overcame this by implementing two new procedures [11]:

- keeping the genotype the same and remapping to new trees using the new grammar;
- keeping the same tree and repairing the genotype to return this tree using the new grammar.

Our approach overcomes the need for remap and repair by adopting a multi-run method, inspired by a similar approach in GP [12]. Not only does it reduce the complexity of implementation but, importantly, it also guarantees valuable information found during the run is not needlessly lost through regular remap or repair.

The many successful applications of GE have been well documented in the past but the motivation in this paper is to further highlight its potential in the area of automatic programming, an area of criticism of GP/GE recently [13]. This paper creates a new module identification technique which contains a more controlled method for selection inspired by biological evolution, further enhancing the capabilities of artificial evolution.

The paper is structured as follows: section 2 gives background on the various attempts at module discovery in GP and GE, section 3 details our new method and sections 4 and 5 give the details and results of our experiments and discussion.

II. Background

The identification, storage and reuse of useful functionality has been a key challenge in GP for many years. The most well known of these methods was pioneered by Koza [14], which he named Automatically Defined Functions (ADFs). The aim was to find reusable, parameterised subprograms during a GP run which can exploit the regularity in a problem to help solve it. They have been shown to greatly increase the power of GP when they are equipped.

Module Acquisition (MA) [15] and Subtree Encapsulation (SE) [16] are other popular variants of this concept. Subtrees of a larger expression are collapsed, their output stored and represented as a node in the program tree. In MA this process is known as *compression* while it is referred to as *encapsulation* in SE. Both highlight how fruitful the addition of modules can be.

Adaptive Representation Through Learning (ARTL) [17] was introduced to overcome the perceived limitations of ADFs and MA. ADFs are randomly modified through crossover and MA randomly select subtrees for compression, which can lead to many redundant or useless modules being created. ARTL tries to find helpful functionality by giving blocks of code a fitness, specified by the user. This enables the system to find good routines.

Run Transferable Libraries (RTL) [18], are a sub-process and exploitation algorithm. They were originally used to pass knowledge fragments from one GP run to another by discovering useful domain functionality [19]. These fragments are stored in a library and reused in subsequent experiments. It has also been used to fight against premature convergence in GP [20]. The elements of the library are simple tree structures, which could be set to be a maximum depth and arity. It is also possible to incorporate some domain knowledge into the library, as they can be seeded with functionality known to be useful [21].

There have previously been attempts to exploit modularity in GE. An early investigation was undertaken by the inventors of GE [22]. In a similar approach to Koza's ADF's, they introduce a parameterised function which is evolved by a grammar alongside and may be used in the main program. The program may recursively call this ADF. This approach showed improvement over standard GE, solving their problem in 100% of runs while standard GE only found the solution in 35% of runs.

Dynamically defined functions were next proposed by [23]. As the name suggests the user does not need to specify any architecture or constraints as they are found during the run. They showed that this approach out performed both GE and GE equipped with ADFs.

The most comprehensive work done on modularity in GE is seen in [11], [24], [25]. They introduce a novel context aware subtree selection procedure, inspired by context aware mutation [26]. A random node is chosen and everything below it encapsulated. Randomly created subtrees replace it in the tree and the tree's fitness recalculated. If the fitness of the tree, including the selected subtree, is better than 70% of the random subtrees which replaced it, it is deemed useful and added to a library. The grammar is augmented with these subtrees, with a maximum of 20 added at a time. This procedure is repeated and the library updated at defined generation steps.

Due to the unique separation between the genotype and the actual executed program, simply adding the new subtrees into the grammar is not a trivial task. The addition of even a single extra non-terminal or production rule can have severe effects on the current genotype-phenotype mapping. To overcome this [11] introduce a *repair* operator which maps the phenotype back to a new genotype using the update grammar. That is to say, this operator returns the identical population of phenotypes, not genotypes, before grammar augmentation occurred.

III. Method

The aim of this work is to investigate a new method of subtree identification in GE and explore its effect on GE's performance across a variety of problem types.

The main issues associated with module encapsulation are the high number of non-useful or potential harmful subtrees which are encapsulated. In GE there is also the harm caused by the addition of modules into the grammar due to the mapping and sizeable effort needed to reduce or mitigate this.

Multi-run subtree encapsulation (MRSE) offers a potential solution to this second problem [12]. MRSE is an extension of SE which encapsulates subtrees across numerous, parallel runs. It splits the run into two parts, the first part searches for subtrees to encapsulate and the second part is a regular GP run with the ability to use the subtrees encapsulated in the first part. An example run may look like:

1) Run GP for 5 generations
2) Encapsulate subtrees
3) Add new subtrees to terminal set
4) Initiate new GP run
5) Run until stopping criteria

This allows the system to best take advantage of the early gains in fitness often seen at the start of evolutionary runs. An approach such as this would suit GE, because a new population is initialised after the modules have been added to terminal set, so the problems associated with augmenting the grammar would disappear as would the need to repair existing individuals. A multi run approach which re-initialises the population means no extra care in grammar construction is needed, the modules are simply added to the grammar as any other terminal would be.

The proposed system, GAME, also goes one step further and attempts to project what traits are likely to be important in future generations. This is achieved by considering the Price equation, a mathematical equation used to describe generational changes in populations [27].

The Price equation states:

$$\Delta z_i \propto cov(w_i, z_i) \tag{1}$$

where Δz is the change in the average frequency a trait present in the population, w_i is the fitness of the individual and z_i is the trait. Covariance is a measure of the relationship between two variables. The sign of the covariance indicates the whether this relationship is positive ($cov > 0$) or negative ($cov < 0$). According to the Price equation, if the covariance between some trait (subtrees in our case) and fitness is positive the amount of that trait in the future generations is expected to increase. Conversely, if the covariance is negative we expect that trait to disappear.

We therefore define fitness of some subtree, or module, x, where w_i is the fitness of each expression and z_i is the number of times the subtree x appears in the expression, as:

$$fitness = \begin{cases} 0 & cov(w_i, z_i) < 0 \\ \frac{Freq\ of\ x}{Total\ Expressions} & cov(w_i, z_i) \geq 0 \end{cases}$$

Ranking trees in this way also leads to possible identification of harmful subtrees. Subtrees with high frequency but very negative covariance may be structures that need removal.

$$\begin{aligned} < exp > ::= & < exp > * < exp > | < exp > / < exp > | \\ & < exp > + < exp > | < exp > - < exp > | \\ & x \mid -1 \mid 1 \end{aligned}$$

Figure 1: Grammar before augmentation

$$\begin{aligned} < exp > ::= & < exp > * < exp > | < exp > / < exp > | \\ & < exp > + < exp > | < exp > - < exp > | \\ & x \mid -1 \mid 1 | < encap > \\ < encap > ::= & Module0 | Module1 .... | Module15 \end{aligned}$$

Figure 2: Grammar after augmentation

That investigation will not be in the scope of this paper, as we focus only on positive covariance for now, but future work discusses some of the implications and advantages of this.

The process of adding the newly encapsulated functionality to the grammar is quite simple. A new non-terminal is created and each rule within this is a subtree. An example basic grammar is seen in Figure 1. A grammar which has been augmented with subtrees is seen in Figure 2.

IV. Experiments

The goal of these experiments was to investigate if this new subtree encapsulation method would improve the performance of a GE run on a selection of problems. The subtree identification and grammar augmentation process is as follows: The population of individuals across 30 parallel runs is first initialised and evolved for 5 generations, as would occur in standard GE. Running 30 parallel experiments and combining them after 5 generations was found to produce fitter and more diverse subtrees that running 1 experiment of 30x500 individuals for 5 generations.

After 5 generations the evolution is stopped and the populations from all 30 runs are amalgamated. The individuals of this new, large population, are broken up and every possible subtree greater than depth 0 isolated. The frequency and the covariance between the frequency of that subtree in an individual and that individual's fitness is calculated. Only subtrees with a positive covariance and frequency above 2 are selected.

To identify functionally identical but morphologically different subtrees, the remaining subtrees are then quickly tested, by considering them as complete individuals, on a sample of 10 test points and subtrees with identical semantics are grouped together. The smallest subtree in this group is then added as the representative of this group. Subtrees which return constants were also omitted from encapsulation. There is much work in constant creation in GE [28], and this is a future avenue for research on different problems, but was omitted from this work.

A maximum of 15 modules, those with the highest fitness, are added to the grammar so as to not bloat the grammar exces-

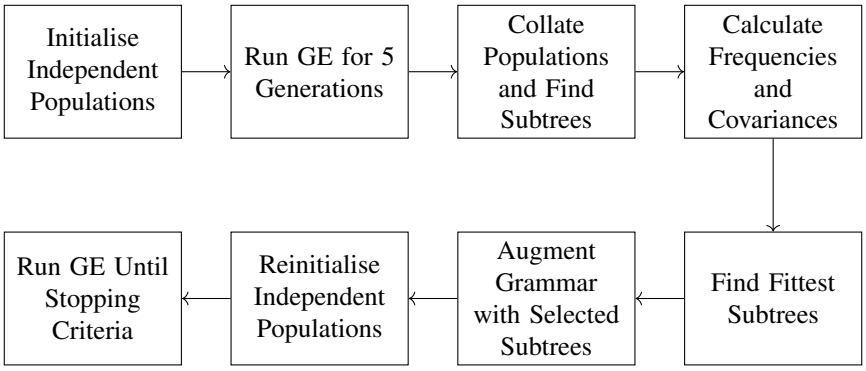


Figure 3: The multi-run grammar augmentation process

sively. The remaining modules are discarded. 15 was chosen to be consistent with [24] but future work will consider different numbers, including dynamically selecting a number based on problem/grammar size or complexity. After the subtrees are added to the grammar the populations are reinitialised in the independent runs and allowed evolve for the remaining 45 generations. An illustration of this scheme is seen in Figure 3.

Seven benchmark problems were chosen from the literature; four symbolic regression problems (Kepler's Law, Quintic Polynomial and two taken from [29], Keijzer-6 and Nguyen-7), one classification (Intertwined Spirals) and two navigation tasks (Santa Fe and Los Altos Ant Trails). The experimental set up for each problem is seen in Table 1. Due to the relative simplicity of the Kepler's Law problem, an additional, random variable was added to the terminal set in order to increase the complexity of the problem. The fitness score for the symbolic regression problems is $1/(Error + 1)$. The maximum fitness for the Santa Fe Ant Trail, Los Altos Ant Trail and Intertwined Spirals is 89, 157 and 194, respectively.

Table I: GE experimental setup

Parameter	**Value**
Runs	30
Total Generations	50
Gens before Encapsulation	5
Population	500 (100 for Ant Trails)
Selection	Tournament
Crossover	0.9 (Effective)
Mutation	0.01
Elitism	Yes
Initialisation	Sensible

V. Results

The results from the experiments are seen in Figures 4-7 and a summary of the results in Table 2. Figures 4-6 show the evolution of the best individual found, averaged per run. GE with GAME was never significantly outperformed by standard GE in any experiment; and standard GE was only able to find a better solution than GE with GAME in one problem, Keijzer-6, although GE with GAME achieved a higher mean final fitness for that problem. Neither approach found the global maximum in any run of that experiment. The Kepler's law problem shows drastic improvement with both the Quintic Polynomial and Los Altos Ant Trail also undergoing significant improvements in performance. The introduction of the modules immediately solved the problem for every run of Kepler's Law, where previously only 23/30 runs had found the solution. GE with GAME found the global optimum for the Quintic Polynomial where standard GE was unable to. The Los Altos Ant Trail saw an increase in the best individual from 105 to 110 with the introduction of the subtrees. A small improvement was seen in the Intertwined spirals problem, with an increase in mean final fitness from 119.7 to 121.1 and best fitness from 127 to 142. However, neither technique was able to find a solution to correctly class every point in the two spirals.

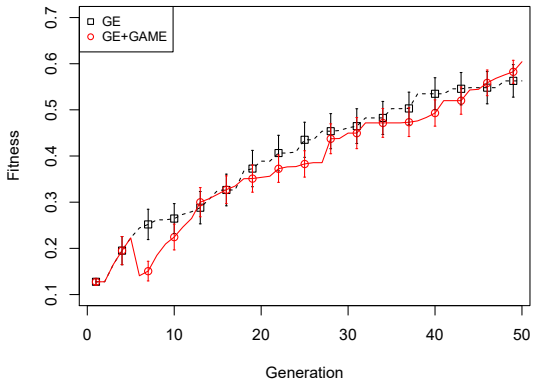
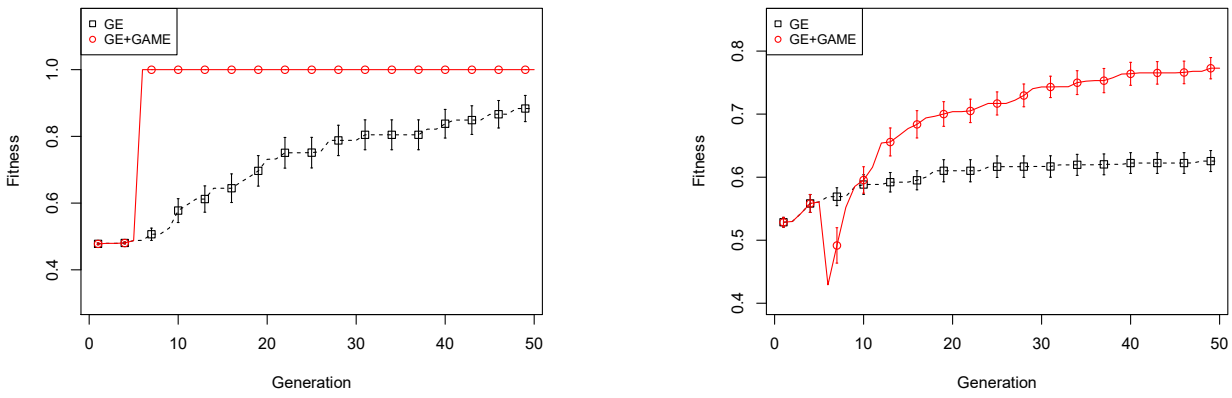


Figure 4: Results of the Keijzer-6 experiment, GE (black squares) and GE with GAME (red circles)

There was no significant difference between GE with grammar augmentation and GE in the Santa Fe Ant Trail: standard GE solved the maze in 18/30 runs, while GE with GAME solved the maze 20/30 times. Mean final fitness was nearly identical, 84.4 and 84.2 respectively.

Keijzer-6, Quintic Polynomial, Nguyen-7 and the Santa Fe problem all exhibit a drop of fitness at generation 5 caused by the grammar augmentation and re-initialisation, a trend



(a) Kepler's Law

(b) Quintic Polynomial

Figure 5: Results of the Kepler's Law and Quintic Polynomial experiments. Each problem was run on GE (black squares) and GE with GAME (red circles).

Table II: Experimental Results

Problem	Experiment	Mean Final Fitness	Found Global Optimum	p-value
Keijzer-6: $\sum_i^x \frac{1}{i}$	GE	0.562	0 (Best 0.817)	0.61
	GE & GAME	0.604	0 (Best 0.808)	
Kepler's Law	GE	0.884	23	**<0.01**
	GE & GAME	1.0	30	
Quintic Polynomial: $x^5 - 2x^3 + x$	GE	0.626	0 (Best 0.808)	**<0.01**
	GE & GAME	0.773	1	
Nguyen-7: $log(x^2 + 1) + log(x + 1)$	GE	0.954	0 (Best 0.986)	0.79
	GE & GAME	0.949	0 (Best 0.987)	
Intertwined Spirals	GE	119.7	0 (Best 127)	0.28
	GE & GAME	121.1	0 (Best 142)	
Santa Fe Ant Trail	GE	84.4	18	0.82
	GE & GAME	84.2	20	
Los Altos Ant Trail	GE	98.8	0 (Best 105)	**<0.01**
	GE & GAME	101.2	0 (Best 110)	

observed in [11], as the population loses all the information it has found when it is remapped to new individuals.

This loss of fitness highlights the problem with the continual augmentation of the grammar with new subtrees, the destruction and loss of information already in the population. However, the new populations and grammar recovered and outperformed or performed as well as standard GE by the final generation.

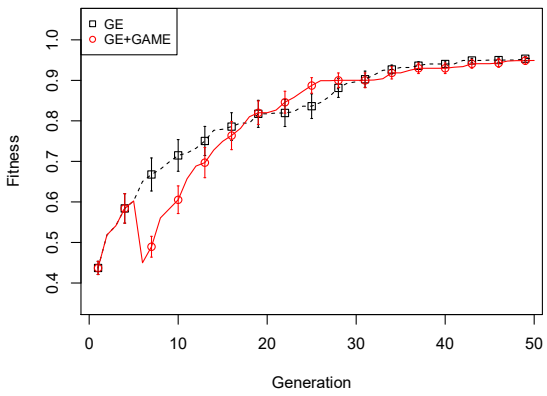
This new technique, adding frequently used subtrees tree with positive covariance to the grammar, can have a very positive effect on the outcome of a GE run. More work is need on examining why it has a drastic positive effect on some problems but a neutral effect on others.

Table 2 shows the summary of the experimental results. The mean final fitness and number of times the global optimum was found is shown for both regular GE and GE with GAME. If neither technique was able to find the global optimum, the best individual fitness from the 30 independent runs was reported. The final column shows the results of significance tests for the two techniques. Results highlighted in bold indicate a statistically significant difference in the techniques. This shows that GE with GAME significantly outperformed standard GE on three of the problems and that there was no difference on the others meaning that at it either helps or makes no difference, but doesn't ever harm GE.

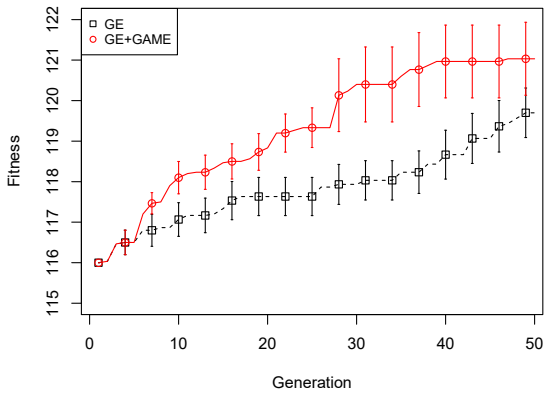
VI. Conclusion & Future Work

Our proposed technique of module identification for GE was investigated and the results highlighted the positive effect it has on finding the problem's solution. It was found to have a very positive effect on three problems while never having a harmful effect on any of the problems examined. Next steps will involve benchmarking our system against other attempts at module encapsulation in GE, although this is a non-trivial task as previous methods are not freely available. Our system, which is integrated into an existing popular GE system, will be made available online upon publication.

The usefulness of the modules chosen in this new approach must be investigated to see if it does indeed reduce the number of useless modules chosen. Only taking subtrees from the highest performing independent runs or individuals seen to be

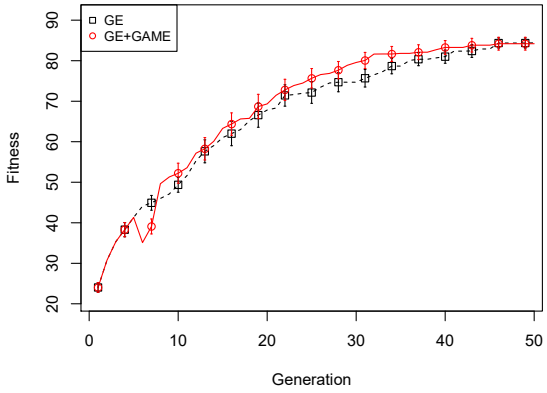


(a) Nguyen-7

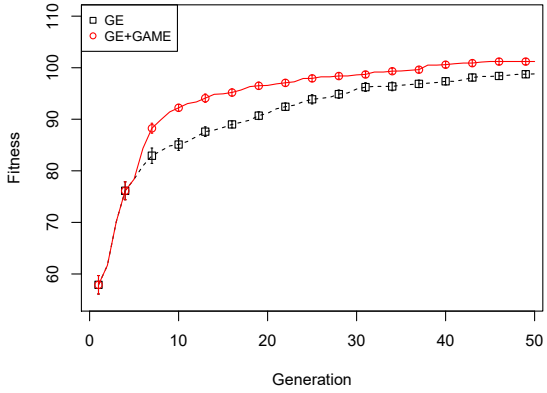


(b) Intertwined Spirals

Figure 6: Results of the Nguyen-7 and Intertwined Spirals experiments. Each problem was run on GE (black squares) and GE with Grammar Augmentation (red circles).



(a) Santa Fe Trail



(b) Los Altos Trail

Figure 7: Results of the two navigation experiments, Santa Fe and Los Altos ant trails. Each problem was run on GE (black squares) and GE with GAME (red circles).

very fit is another possibility. GE can produce programs in any arbitrary language, looking into which types of representation are best suited to this new approach is also an avenue for research.

Once modules are added to the grammar they are unable to be altered. Adding the ability for these to be mutated, or indeed removed, in future generations or multi-run stages is another area to be investigated.

This paper attempted identify modules which were useful in finding the optimal solution for the problems considered, not in identifying modules which are reused. Running this technique on problems which require reuse, such as program synthesis, rather than symbolic regression or classification is another interesting avenue for future work.

Adding modules while at the same time pruning sub-trees/terminals from the grammar would no longer fall in the domain of module acquisition and instead be somewhere between automatic grammar design and encapsulation [30]. Removing terminals from the grammar would not help guide the search but would instead reshape the search space. This comes with many other considerations and will be outside the scope of this paper, but does leave open an intriguing possibility.

It is also possible that this technique could be used in conjunction with a probabilistic CFG to tune the probabilities or in conjunction with an attribute grammar to tune or create new attributes.

ACKNOWLEDGMENT

The authors thank the anonymous referees for their time, comments and helpful suggestions. This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

REFERENCES

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[2] T. Back, D. B. Fogel, and Z. Michalewicz, Eds., *Handbook of Evolutionary Computation*, 1st ed. Bristol, UK, UK: IOP Publishing Ltd., 1997.

[3] G. Gerules and C. Janikow, "A survey of modularity in genetic programming," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 5034–5043.

[4] T. Helmuth, N. F. McPhee, E. Pantridge, and L. Spector, "Improving generalization of evolved programs through automatic simplification," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 937–944. [Online]. Available: http://doi.acm.org/10.1145/3071178.3071330

[5] D. Howard and M. A. Edwards, "Explainable ai: The promise of genetic programming multi-run subtree encapsulation," in *2018 International Conference on Machine Learning and Data Engineering (iCMLDE)*. IEEE, 2018, pp. 158–159.

[6] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[7] C. Ryan, J. J. Collins, and M. O. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *European Conference on Genetic Programming*. Springer, 1998, pp. 83–96.

[8] C. Ryan, M. O'Neill, and J. Collins, *Handbook of Grammatical Evolution*. Springer, 2018.

[9] M. O'Neill and A. Brabazon, "Grammatical swarm: The generation of programs by social programming," *Natural Computing*, vol. 5, no. 4, pp. 443–462, 2006.

[10] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico, "Crossover in grammatical evolution," *Genetic Programming and Evolvable Machines*, vol. 4, no. 1, pp. 67–93, Mar 2003. [Online]. Available: https://doi.org/10.1023/A:1021877127167

[11] J. M. Swafford, E. Hemberg, M. O'Neill, M. Nicolau, and A. Brabazon, "A non-destructive grammar modification approach to modularity in grammatical evolution," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 2011, pp. 1411–1418.

[12] D. Howard, "Modularization by multi-run frequency driven subtree encapsulation," in *Genetic Programming Theory and Practice*. Springer, 2003, pp. 155–171.

[13] M. O'Neill and L. Spector, "Automatic programming: The open issue?" *Genetic Programming and Evolvable Machines*, Sep 2019. [Online]. Available: https://doi.org/10.1007/s10710-019-09364-2

[14] J. R. Koza, "Genetic programming 2: Automatic discovery of reusable subprograms," *Cambridge, MA, USA*, vol. 13, no. 8, p. 32, 1994.

[15] P. J. Angeline and J. B. Pollack, "The evolutionary induction of subroutines," in *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*. Bloomington, Indiana, USA: Lawrence Erlbaum, 1992, pp. 236–241. [Online]. Available: http://www.demo.cs.brandeis.edu/papers/glib92.pdf

[16] S. C. Roberts, D. Howard, and J. R. Koza, "Evolving modules in genetic programming by subtree encapsulation," in *Genetic Programming*, J. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 160–175.

[17] J. P. Rosca and D. H. Ballard, "Learning by adapting representations in genetic programming," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. IEEE, 1994, pp. 407–412.

[18] M. Keijzer, C. Ryan, and M. Cattolico, "Run transferable libraries — learning functional bias in problem domains," in *Genetic and Evolutionary Computation – GECCO 2004*, K. Deb, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 531–542.

[19] C. Ryan, M. Keijzer, and M. Cattolico, *Favourable Biasing of Function Sets Using Run Transferable Libraries*. Boston, MA: Springer US, 2005, pp. 103–120. [Online]. Available: https://doi.org/10.1007/0-387-23254-0_7

[20] G. Murphy and C. Ryan, *Manipulation of Convergence in Evolutionary Systems*. Boston, MA: Springer US, 2008, pp. 33–52. [Online]. Available: https://doi.org/10.1007/978-0-387-76308-8_3

[21] ——, "Seeding methods for run transferable libraries," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1755–1755. [Online]. Available: http://doi.acm.org/10.1145/1276958.1277305

[22] M. O'Neill and C. Ryan, "Grammar based function definition in grammatical evolution." in *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2000, pp. 485–490.

[23] R. Harper and A. Blair, "Dynamically defined functions in grammatical evolution," in *2006 IEEE International Conference on Evolutionary Computation*. IEEE, 2006, pp. 2638–2645.

[24] J. M. Swafford, M. O'Neill, M. Nicolau, and A. Brabazon, "Exploring grammatical modification with modules in grammatical evolution," in *European Conference on Genetic Programming*. Springer, 2011, pp. 310–321.

[25] J. Swafford, M. Nicolau, E. Hemberg, M. O'Neill, and A. Brabazon, "Comparing methods for module identification in grammatical evolution," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM, 2012, pp. 823–830.

[26] H. Majeed and C. Ryan, "Context-aware mutation: a modular, context aware mutation operator for genetic programming," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1651–1658.

[27] G. R. Price, "Fisher's 'fundamental theorem' made clear," *Annals of human genetics*, vol. 36, no. 2, pp. 129–140, 1972.

[28] R. M. A. Azad and C. Ryan, *Comparing Methods to Creating Constants in Grammatical Evolution*. Cham: Springer International Publishing, 2018, pp. 245–262. [Online]. Available: https://doi.org/10.1007/978-3-319-78717-6_10

[29] D. R. White, J. McDermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke, "Better gp benchmarks: community survey results and proposals," *Genetic Programming and Evolvable Machines*, vol. 14, no. 1, pp. 3–29, Mar 2013. [Online]. Available: https://doi.org/10.1007/s10710-012-9177-2

[30] R. M. A. Azad and C. Ryan, "An examination of simultaneous evolution of grammars and solutions," in *Genetic Programming Theory and Practice III*. Springer, 2006, pp. 141–158.