A Comparative Study of Genetic Programming and Grammatical Evolution for Evolving Data Structures

Kevin Igwe
School of Mathematics, Statistics and Computer Science
University of KwaZulu-Natal
Pietermaritzburg, South Africa
igwekevin@gmail.com

Nelishia Pillay
School of Mathematics, Statistics and Computer Science
University of KwaZulu-Natal
Pietermaritzburg, South Africa
pillayn32@ukzn.ac.za

***Abstract*—The research presented in the paper forms part of a larger initiative aimed at automatic algorithm induction using machine learning. This paper compares the performance of two machine learning techniques, namely, genetic programming and a variation of genetic programming, grammatical evolution, for automatic algorithm induction. The application domain used to evaluate both the approaches is the induction of data structure algorithms. Genetic programming is an evolutionary algorithm that searches a program space for an algorithm/program which when executed will provide a solution to the problem at hand. Grammatical evolution is a variation of genetic programming which provides a more flexible encoding, thereby eliminating the sufficiency and closure requirement imposed by genetic programming. The paper firstly extends previous work on genetic programming for evolving data structures, providing an alternative genetic programming solution to the problem. A grammatical evolution solution to the problem is then presented. This is the first application of grammatical evolution to this domain and for the simultaneous induction of algorithms. The performance of these approaches in inducing algorithms for the stack and queue data structures are compared.**

Keywords—algorithm induction; genetic programming; grammatical evolution; automatic programming

I. INTRODUCTION

The paper reports on a study that forms part of a project investigating automatic algorithm induction and design using machine learning. One of the areas researched as part of this project is automatic algorithm induction as a means of automatic programming. Genetic programming, a machine learning technique for solving optimization problems, appears to be apt for this purpose. Genetic programming searches a program space for a program, which when executed will produce a solution to the problem at hand[1]. Each program is generally represented as a parse tree. Genetic programming has been successfully applied to various domains including data mining, natural language processing, image processing and electronic circuit design [2].

There have been various attempts at using genetic programming for automatic programming. In [3] genetic programming is used to evolve algorithms according to the imperative programming paradigm, using memory, iteration and modularization. Algorithms are evolved in an internal representation language to facilitate language independence and can be converted into any procedural programming language. As the field of genetic programming advanced, researchers started looking to good programming practices to improve the scalability and problem solving ability of genetic programming. One such practice is object-oriented programming which led to the extension of genetic programming to object-oriented genetic programming (OOGP) [4-7]. OOGP evolves object-oriented programs. This work has essentially focused on the induction of algorithms for method implementation rather than the evolution of classes and interfaces. Methods for a class are generally evolved simultaneously.

OOGP has also been used for purposes of automatic programming [8-10]. Bruce [8] compares the sequential and simultaneous induction of methods to evolve object-oriented programs. Each method is an automatically defined function [11] and all methods are stored in indexed memory. The proposed approach for OOGP is evaluated in the domain of data structure algorithm induction. A similar approach is taken by Langdon [9]. This study researches the induction of both methods for classes and programs using instances of the classes. The approach is also tested for the evolution of data structure algorithms as well as solution algorithms for problems requiring the use of the evolved data structures. In [10] a rule-based expert system is used to induce an object-oriented design (OOD) from a program specification. The OOD forms input to a genetic programming component which evolves the methods for the program sequentially, allowing function calls between methods.

More recent studies in the area of OOGP include initial investigations into grammar-based genetic programming for the evolution of object-oriented programs [12] and a combination of OOGP and linear genetic programming [13]. Grammatical evolution is a variation of genetic programming which aims at providing a more flexible encoding of programs thereby allowing for programs to be generated in any language [14]. Grammatical evolution (GE) essentially evolves a population of binary strings which represent programs. The execution of a program involves converting the binary string into an integer which is then mapped onto a grammar, resulting in a production rule of the grammar being executed [14]. We hypothesize that grammatical evolution has the potential to contribute to the domain of automatic object oriented programming. To the authors' knowledge there has been no previous work into grammatical evolution for object-oriented program induction or the simultaneous induction of algorithms. This study compares both OOGP and GE for the simultaneous induction of algorithms. The application domain for evaluation of the proposed approaches is the induction of algorithms for data structures. This domain has been chosen as it is a problem that has previously been used for the evaluation of OOGP performance and has proven to be suitable to test such approaches. Furthermore, this will also allow for a comparison of the approaches proposed in this study to previous methods used for automatic algorithm induction. Hence the research presented makes the following contributions:

1. A variation of OOGP, namely, GOOGP which uses a greedy method to create the initial population.

2. A grammatical evolution solution to automatic object-oriented programming.

3. An evaluation of GE for simultaneous induction of algorithms.

4. A comparison of the performance of the three approaches for automatic object-oriented programming.

The following section presents an OOGP approach for automatic algorithm induction. Section III proposes a grammatical evolution solution to the problem. The experimental setup used to evaluate the performance of both the approaches is presented in section IV. Section V discusses the performance of both approaches and provides an empirical comparison with previous work. A summary of the findings of this research and future extensions of this work are presented in section VI.

II. OBJECT ORIENTED GENETIC PROGRAMMING (OOGP) APPROACH FOR AUTOMATIC PROGRAMMING

This section describes the genetic programming approach implemented for automatic programming. The generational genetic programming algorithm in Fig. 1 is used. The algorithm begins by creating an initial population which is iteratively refined by means of evaluation, selection and regeneration until the termination criteria are met. These processes are described in the following sections. The algorithm is terminated when a solution chromosome is found or the maximum number of generations has been reached.

A. Initial Population Generation

As in the studies conducted by Bruce [8] and Langdon [9], each element of the population, i.e. a chromosome, represents a class and is comprised of parse trees, one for each method of the class. Each tree is a gene in the chromosome. The algorithms are generated by an internal representation language which is language independent. This allows for the evolved algorithms to be converted to any programming language. An example is illustrated in Fig. 2.

```
Create an initial population
Repeat
  Evaluate the population
  Selection parent
  Apply genetic operators
Until the termination criterion is met
```

Fig. 1. Generational genetic programming algorithm

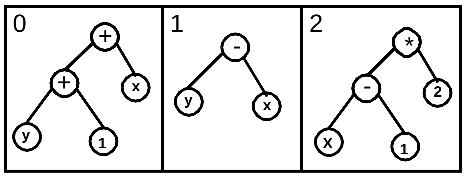

Fig. 2. Example of a chromosome

The class has three methods. Each chromosome in the population is indexed memory, with each of the trees, i.e. genes, stored at an index. The internal representation language is defined by the function and terminal set. Each parse tree is created by randomly choosing elements from the function and terminal sets until a preset maximum depth is reached. The grow method [1] is used for this purpose. The function set used for each problem is a subset of the following:

- Arithmetic operators: +, -, *, / which perform the standard arithmetic operations. The division operator is a protected operator which returns a value of 1 if the denominator is 0.
- Conditional operator: *if* which performs the function of an *if-then-else* operator.
- Arithmetic logical operators: ==, !=, <, >, <=, >= which perform the standard arithmetic logical operations and are used to create the subtree representing the condition of the *if* operator.
- Indexed memory operators: Indexed memory is maintained which each program can write to or read from. The *write* and *read* operators are used to access indexed memory. The *write* operator takes two arguments, one the content to be written and the second the index in memory to which it should be written. The *read* operator takes a single argument, namely the index from which the content should be read.
- Named memory operators - A single named memory location, *aux*, is maintained which a program can use as a temporary memory location. Two operators, *set_aux* and *dec_aux*, defined in [9] are used to access the memory location. The operator *set_aux* is used to write to this memory location. It takes one argument, which is the content to be written to the location. The operator *dec_aux* decrements the value in the memory location by one.
- Multiple statement operators - *block2* and *block3* are used to combine programming statements, namely, two and three statements respectively. Both these operators return the value that the last argument evaluates to. The operator *fblockn* combines *n* programming statements but returns the value that the first argument evaluates to.
- Iteration operator - The *for* operator defined in [15] is used to cater for iteration. The operator takes three arguments. The first two arguments represent the bounds of the loop and the third argument represents the body of the loop. The operator evaluates to the value of the body on the last iteration of the loop.

The terminal set is a subset of:

- A named memory location: *aux* is a single memory location which can be used for temporary storage and serves the same function as a variable in programming.
- Variables representing input to the problem, e.g. *i* representing the value to be stored in the data structure.
- Constants: 0 and 1.

Two methods are tested for creating the initial population. The first is the standard approach adopted in genetic programming where each element of the population is randomly created. We will refer to the genetic programming algorithm using the standard approach as OOGP in the sections that follow. The second is a greedy approach which creates each chromosome in the population as follows:

- A population of *m* parse trees is randomly created for each gene.
- The population is evaluated using the process described in section B below to determine the fitness of each tree.
- The fittest tree in the population is stored as the gene for the chromosome.

The genetic programming algorithm employing the greedy approach will be referred to as GOOGP. The following section describes fitness evaluation and selection.

B. *Fitness Evaluation and Selection*

The fitness is maximized, thus a fitter chromosome is one with a higher fitness value. For each run 15 fitness cases are randomly generated. Each fitness case is a stack of length between 1 and 15. The elements of the stack are integer values in the range 1 to 99. Each method in the chromosome is applied to the 15 fitness cases. A set of problem dependent criteria that must be met by each method is defined and the method is scored on the number of criteria it has met. An example of this is listed in Table 1 for the stack data structure. The class has five methods, namely, *makeNull()*, *peek()*, *push()*, *pop()* and *empty()*. The maximum score that each of the methods can attain respectively is 1, 4, 4, 3 and 3 giving a maximum fitness of 15 per fitness case and 225 over all 15 fitness cases. However, when selecting parents the fitness is calculated relative to the best performing method in the current population. The best score obtained in the current population for each of the methods is stored (*B1*, *B2*, *B3*, *B4*, *B5*). The fitness is calculated relative to the best. For example, suppose that fitness scores for a chromosome are *F1* to *F5* for each of the five methods. The fitness of a potential parent is calculated using the following formula:

$$Relative\ Fitness = \Sigma\ (Fi/Bi)*100 \quad i=1, ..., no.\ of\ methods$$

Tournament selection [1] is used to then choose the parent. This selection method chooses a tournament of *t* chromosomes from the population and calculates the relative fitness. The chromosome with the highest relative fitness is returned as a parent. Selection is with replacement so a chromosome can play the role of a parent more than once.

TABLE 1. EXAMPLE PROBLEM SPECIFIC CRITERIA FOR FITNESS EVALUATION

Method	Criteria
makeNull()	• Stack pointer must be set to -1
peek()	• No change in pointer value. • Elements on stack should not be altered. • The value returned must be the topmost element of the stack. • Only one value must be returned.
push()	• Pointer must be updated correctly. • Elements on the stack should not be altered. • The pushed value must be at the top of the stack. • The value must be pushed on the stack only once.
pop()	• Pointer must be updated correctly • Elements on the stack should not be altered. • The correct value must be returned.
empty()	• No change in pointer value. • Elements on the stack should not be altered. • The correct position of the pointer must be returned.

C. *Regeneration*

The crossover operator performs two phases of crossover, namely, external crossover followed by internal crossover. External crossover performs uniform crossover used by genetic algorithms [16]. Two chromosomes are selected using tournament selection and genes are swapped between the chromosomes if the randomly generated probability in the range 1 to 100 is less than the preset probability.

For example suppose that the selected parents are $G_{11}G_{12}G_{13}G_{14}$ and $G_{21}G_{22}G_{23}G_{24}$. Each chromosome is comprised of four genes. Each gene G_{ij} represents a parse tree. Given that the preset probability is 60% and the randomly generated probabilities for each gene are 34%, 75%, 80%, 25% respectively, the resulting offspring are $G_{21}G_{12}G_{13}G_{24}$ and $G_{11}G_{22}G_{23}G_{14}$.

Once external crossover is applied a number is randomly generated in the range 1 to 100 again. If this number is less than the preset probability of internal crossover (internal crossover probability) this operator is applied as follows. A chromosome index is randomly chosen. The standard genetic programming crossover operator [1] is applied to the parse trees at the selected index in both parents. This operator randomly selects crossover points in each of the parse trees, and the subtrees rooted at these points are swapped. The resulting trees replace the parents in the chromosomes. The fitter of the two offspring forms part of the next generation.

III. GRAMMATICAL EVOLUTION APPROACH FOR AUTOMATIC PROGRAMMING

This section describes the grammatical evolution approach for automatic programming. The main difference between this and the OOGP approach is that the each chromosome is a binary string. The space of binary strings is then mapped onto an integer space which is in turn mapped onto a grammar which represents the programming statements. The algorithm implemented for evolution is the generational algorithm in Figure 1. The GE approach uses the same termination criteria and processes as the GP approach for fitness evaluation and selection described in section II. Initial population generation and regeneration for the GE are described below.

A. Initial Population Generation

GE requires each element of the population to be represented as a binary string. In the OOGP approach each chromosome is represented as indexed memory with each gene in the memory a parse tree corresponding to a method of the class. A similar approach is taken in GE. Each chromosome is again indexed memory, but each gene is a binary string representing a method of the class. Each binary string is called a codon and is composed of *n* alleles of length *m*. An example is illustrated in Fig. 3. The chromosome contains four binary strings, each representing one of four methods for the class. Each binary string is a codon of length 3, i.e. contains 3 alleles. Each allele is of length 8. Each codon represents a program. In order to execute the program represented by each codon, each of the alleles in the codon are firstly converted to denary. For example, the first codon will be converted to 15, 5, 65. The denary values are mapped onto a grammar defining the valid programming statements for the class. For example the grammar used for inducing the methods of the stack class are illustrated in Fig. 4. Each denary value is then mapped to a production rule in the grammar for the method thereby converting the codon to a parse tree representing the program. For example suppose that the grammar start symbol is <stmts> and production rules for <stmts> are:

<stmts> :: <stmt> (0)

<stmts>:: <stmt>;<stmts> (1)

There are two production rules for the non-terminal <stmts>. To decide which production rule to apply the modulus of the denary value and the number of production rules is taken. This process is continued to convert the codon into a parse tree representing a program. If there are still non-terminal variables that need to be expanded once the codon has been processed, the process begins at the beginning of the codon again with the first allele. The construction of the parse tree ends when all non-terminals have been expanded. In order to prevent cyclic calls to non-terminals, a limit is set on the number of times a non-terminal can be called in the program.

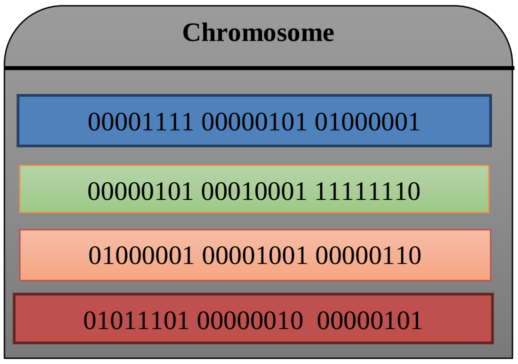

Fig. 3. Example of a chromosome

$$
\begin{aligned}
&< \text{stmts} > :: = < \mathit{stmt} >; < \text{stmts} >; \mid < \mathit{stmt} >; \\
&< \mathit{stmt} > :: = < \mathit{write}\ (\mathit{expr}, \mathit{expr}) > \mid < \text{read}\ (\text{expr}) > \\
&\qquad\qquad\qquad \mid < \mathit{set\_aux}\ (\mathit{expr}) > \mid < \mathit{expr} > \\
&< \mathit{expr} > :: = \ < + >< \mathit{var} >< \mathit{var} > \mid < - >< \mathit{var} > \\
&\qquad\qquad\qquad < \mathit{var} > \mid < \mathit{var} > \\
&< \mathit{var} > :: = \ 0 \mid 1 \mid N \mid \mathit{aux}
\end{aligned}
$$

Fig. 4. Grammar for the Stack class

B. Regeneration

The crossover and mutation operators are used for regeneration. The crossover operator is essentially the same crossover operator defined for OOGP, but is applied to bits in the codon instead of parse trees. The external crossover operators swap bits between parents if the probability for the bit is less than the preset probability. As in the case of OOGP internal crossover is only applied if the random number generated is less than the preset probability. Internal crossover is essentially one-point crossover [16]. A crossover point is chosen in the parents and codons are crossed over at that point to produce two offspring. As in the case of OOGP the fitter of the two offspring is returned as the result of the operation.

The mutation operator is applied to the offspring created by crossover. A mutation probability and bit flip probability is set for mutation. The values of these parameters are problem dependent. A random number between 1 and 100 is generated. If this value is less than the mutation probability, mutation is performed. A random number between 1 and 100 is generated for each bit in the codon. If this number is less than the bit flip probability the bit is flipped, i.e. if the bit is 0 it becomes 1 and vice versa.

IV. EXPERIMENTAL SETUP

This section describes the experimental setup for testing OOGP, GOOGP and GE in algorithm induction for automatic programming. The first section describes the problem domain that the approaches were tested on. Section B presents the parameter values used for the approaches and section C technical specifications.

A. Problem Domain

Data structure algorithm induction was used to evaluate the approaches. Each of the approaches was tested on the simultaneous evolution of methods for the stack and queue data structures. Table 2 lists the methods that need to be induced for the array-based stack ADT and Table 3 those for the array-based queue ADT as defined in [9].

B. Parameter Values for Approaches

The function and terminal sets used for the stack and queue ADTs are listed in Table 4. The parameter values used for OOGP, GOOGP and GE were obtained empirically by performing trial runs. These are listed in Table 5 for GP and Table 6 for GE. Various values were tested for each of the parameters and those performing the best were selected. Due to the stochastic nature of the approaches tested, 30 runs, each with a different random number generator seed, was performed for the stack and queue ADTs.

TABLE 2. METHODS FOR THE STACK ADT

Methods	Function
makeNull()	Sets the pointer to the stack to -1. The return value is ignored.
push()	Push an integer onto the stack. Return value is ignored.
peek()	Returns the topmost value on the stack.
pop()	Returns the topmost value in the stack, removes the value from the stack and decrements the *aux* by 1.
empty()	Returns an integer less than zero if the stack is empty, otherwise it returns an integer greater or equal to zero.

TABLE 3. METHODS FOR THE QUEUE ADT

Methods	Function
makeNull()	Sets the pointer to the stack to -1. The return value is ignored.
enqueue()	Enqueue an integer. Return value is ignored.
front()	Returns the value in the front of the queue.
dequeue()	Returns the value in the front of the queue, removes the value from the queue and decrements the pointer by 1.
empty()	Returns an integer less than zero if the queue is empty, otherwise it returns an integer greater or equal to zero

TABLE 4. FUNCTION AND TERMINAL SET

Data Structure	Function Set	Terminal Set
Stack	+, -, *, /, if, >=, <=, ==, !=, *block2, block3, write, read, set_aux*	*i* (integer to push on stack), 0, 1, *aux*
Queue	+, -, *, /, if, >=, <=, ==, !=, *fblockn, write, read, for, set_aux, dec_aux*	*i* (integer to enqueue), 0, 1, *aux*

TABLE 5. PARAMETER VALUES FOR OOGP AND GOOGP

Parameter	Stack	Queue
Population size	100	500
Maximum depth range	3-5	3-5
Tournament Size	2	2
External crossover probability	50	50
Internal crossover probability	50	50
Maximum offspring depth range	4-10	4-10
Number of Generations	50	50
Population size (n) for GOOGP	500	500

TABLE 6. PARAMETER VALUES FOR GE

Parameter	Stack	Queue
Population size	500	500
Codon length	10	10
Allele length	8	8
Tournament size	4	4
Mutation probability	30	40
Bit flip probability	70	70
External crossover probability	50	80
Internal crossover probability	70	50
Number of Generations	100	100

C. *Technical Specifications*

The system was implemented in Java using Netbeans IDE 7.2.1 with JDK 1.7.2_25. Simulations were run on an Intel Core 3.1GHz machine with 8192 MB of RAM.

V. RESULTS AND DISCUSSION

This section discusses the performance of OOGP, GOOGP and GE in inducing methods for the stack and queue ADTs. The performance of the approaches is evaluated in terms of their ability in evolving solution methods for each ADT. This is reported as a success rate. Thirty runs have been performed for each approach for each ADT. The success rate is the number of the 30 runs that have produced a chromosome with solution algorithms for all methods of the class. Table 7, Table 8 and Table 9 lists the success rate, average fitness and the average runtimes (in milliseconds) for each of the approaches for both the ADTs. The best fitness that can be obtained is 225. For 29 of the 30 runs conducted for both GE and GOOGP solution chromosomes were found with a fitness of 225 for the stack ADT. OOGP did not perform as well, producing a solution chromosome on only one of the runs with an average fitness of 197.47 over the 30 runs for the stack ADT. GE was able to produce solutions quicker than GOOGP with lower average runtimes. Hypothesis tests were conducted to test statistical significance of these results. Three hypotheses were tested:

- Hypothesis 1: GOOGP performs better than OOGP.
- Hypothesis 2: GE performs better than OOGP.
- Hypothesis 3: Runtimes of GE are better than GOOGP.

The hypotheses were tested at 5%, 10% and 15% levels of significance and were found to be significant at all levels. The Z values are listed in Table 10.

GE performs better than both GOOGP and OOGP in evolving the queue data structure. GE finds solution chromosomes on all 30 runs, while GOOGP finds solutions on 24 of the 30 runs and OOGP was not able to find a solution chromosome.

TABLE 7. SUCCESS RATES

ADT	OOGP	GOOGP	GE
Stack	1	29	29
Queue	0	24	30

TABLE 8. AVERAGE FITNESS

ADT	OOGP	GOOGP	GE
Stack	197.47	224.50	224.57
Queue	195.37	222.4	225

TABLE 9. RUNTIMES

ADT	OOGP	GOOGP	GE
Stack	1623	16326	4287
Queue	21501.33	157874.33	7127.87

TABLE 10. Z-VALUES FOR HYPOTHESIS TESTS: STACK ADT

Hypothesis	Z-Value
Hypothesis 1	13.81
Hypothesis 2	13.96
Hypothesis 3	15.42

Hypothesis tests were conducted to test the statistical significance of these results. The hypotheses tested is that GE performs better than OOGP and GE performs better than GOOGP in evolving the queue data structure. The hypotheses were found to be significant at all levels of significance with Z values of 2.72 and 21.84 respectively.

GOOGP and GE have performed comparatively to the OOGP approaches employed by Bruce [8] and Langdon [9] in evolving the stack data structure with GE outperforming GP. In [8] 20 runs were performed and one solution that correctly induced all the five methods in the stack class was found. In [9] 4 solution chromosomes were found on the 60 runs performed for the stack class. Similar performance was attained for the evolution of the queue data structure. A chromosome that could evolve all queue methods correctly could not be found by the OOGP employed in [8]. It was however reported that 2 of the generated individuals correctly induced 3 of the 5 required methods. In [9] the OOGP approach found a solution on one of the 379 runs performed.

Fig. 5 illustrates one of the solutions found for the stack ADT using OOGP. This is a push down stack which increments the stack pointer when an element is pushed onto the stack and decrements the pointer when an element is popped. Introns are redundant code which genetic programming is known to generate as part of solution programs [2]. The introns have been removed from the solution programs displayed in Fig. 5 to improve the readability of programs. The named memory location *aux* is used as a stack pointer. The solution for the *makeNull* method sets the stack pointer to -1. The evolved method for *push* firstly increments the stack pointer by 1 and then writes the element *i* to the position in indexed memory pointed to by the stack pointer. The *set_aux* operator evaluates to zero so the method returns the value pushed onto the stack. The method evolved for *peek* reads the value currently pointed to by the stack pointer. The *pop* method firstly reads the value in memory indexed by the stack pointer and then decrements the stack pointer by 1. The *empty* method returns the value of the stack pointer, if this value is negative it means the stack is empty.

An example of an evolved queue ADT evolved by GE is depicted in Fig 6. As in the case of the stack solution in Fig. 5, the named memory location *aux* is used as a queue pointer to the queue which is stored in indexed memory.

Method	**Solution**
makeNull()	set_aux(0-1)
push()	set_aux(1+aux) + write(i,aux)
peek()	read(aux)
pop()	read(aux) + set_aux(aux -1)
empty()	aux

Fig. 5. Stack solution evolved by OOGP

Method	**Solution**
makeNull()	set_aux(0-1)
enqueue()	set_aux(1+aux) write(i,aux)
front()	read(0)
dequeue()	for(aux, dec_aux, write(lvar5,cvar5))
empty()	aux

Fig. 6. Queue solution evolved by GE

The *makeNull* method sets the queue pointer to -1 to indicate that the queue is empty. The evolved *enqueue* method firstly increments the queue pointer. The integer value is then written to the memory location pointed to by the queue pointer. The *front* method reads the element stored at index 0 in the indexed memory. The *dequeue* method uses the *for* operator to achieve its aim. The first argument of the *for* operator is the queue pointer stored in *aux*. The second argument decrements the queue pointer by 1 and returns a value of zero. The *for* operator implemented in this study maintains a counter variable (*cvar*) and an iteration variable (*ivar*) for each *for* operator instance [15]. The counter variable stores the counter value for the iteration. For example if a *for* loop beings at 1 and ends at 3, then the value of *cvar* will be 1, 2 and 3 respectively on each iteration. The iteration value stores what the body has evaluated to on the last iteration. The iteration variable is given an initial value of 0.

VI. Conclusion and Future Work

The aim of this study was to compare the performance of object-oriented genetic programming and grammatical evolution for the automatic induction of algorithms. Three approaches, namely, object-oriented genetic programming (OOGP), object-oriented genetic programming using a greedy method to create the initial population (GOOGP) and grammatical evolution (GE) were implemented to evolve the stack and queue data structures. Grammatical evolution was found to perform better than both GOOGP and OOGP in the simultaneous induction of algorithms with lower runtimes than GOOGP. GOOGP produced much better results than OOGP. GE and GOOGP were also found to perform better than previous OOGP approaches evaluated for the evolution of the stack and queue data structures. Future work will include a further analysis of the results to identify the theoretical justification for the performance of the three approaches. The application of GE to additional automatic object-oriented programming problems, including the evolution of programs that use instances of classes, will also be investigated as future extensions of this work.

Acknowledgment

This work is based on the research supported in part by the National Research Foundation of South Africa for the Grant CSUR13091742778. Any opinion, finding and conclusion or recommendation expressed in this material is that of the author(s) and the NRF does not accept any liability in this regard.

References

[1] J. R. Koza, Genetic Programming I : On the Programming of Computers by Means of Natural Selection - John R. Koza, MIT Press, 1992.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, Genetic Programming - AnIntroduction - On the Automatic Evolution of Computer Programs and Its Applications, Morgan Kaufmann Publishers, Inc., 1998.

[3] K. Igwe and N. Pillay, “Automatic Programming Using Genetic Programming,” in Proceedings of the World Congress on Information and Communication Technologies., Hanoi, Vietnam., pp. 339 – 344, December 2013.

[4] R. Abbott, “Object-oriented GeneticPprogramming, An Initial Implementation,” in proceedings of the International Conference on

Machine Learning: Models, Technologies and Applications, pp. 26-30, 2003.

[5] S. M. Lucas, “Exploiting Reflection in Object Oriented Genetic Programming,” in Genetic Programming, Lecture Notes in Computer Science, Vol. 3003, Springer, pp. 369–378, 2004.

[6] A. Agapitos and S. M. Lucas, “Learning Recursive Functions with Object Oriented Genetic Programming,” in Genetic Programming, Lecture Notes in Computer Science, Vol. 3905, Springer, pp. 166–177, 2006.

[7] A. Agapitos and S. M. Lucas, “Evolving Modular Recursive Sorting Algorithms,” in Genetic Programming, Lecture Notes in Computer Science, Vol. 4445, Springer, pp. 301–310, 2007.

[8] W. S. Bruce, “The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs,” Phd Thesis, Nova Southeastern University, 1995.

[9] W. B. Langdon, Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming! Kluwer Academic Publishers, 1998.

[10] N. Pillay and C. K. Chalmers, “A Hybrid Approach to Automatic Programming for the Object-Oriented Programming Paradigm,” in proceedings of the 2007 conference of the South African Institute of Computer Scientists and Information Technologists, pp. 116–124, 2007.

[11] J. R. Koza, Genetic Programming II, Automatic Discovery of Reusable Programs, MIT Press, 1994.

[12] Y. Oppacher, F. Oppacher, and D. Deugo, “Evolving Java Objects Using a Grammar-Based approach,” in proceedings of the conference on Genetic and Evolutionary Computation (GECCO 2009), pp. 1891–1892, 2009.

[13] M. R. Medland, K. R. Harrison, and B. Ombuki-Berman, “Incorporating Expert Knowledge in Object-Oriented Genetic Programming,” in proceedings of the conference on Genetic and Evolutionary Computation (GECCO 2014), pp. 145–146, 2014.

[14] M. O'Neil and C. Ryan, Grammatical Evolution, Evolutionary Automatic Programming in an Arbitrary Language, Kluwer Academic Publishers, 2003.

[15] N. Pillay, “Evolving Solutions to ASCII Graphics Programming Problems in Intelligent Programming Tutors,” in proceedings of the International Conference on Applied Artificial Intelligence (ICAAI’2003), pp. 236–243, 2003,.

[16] D. Goldberg D, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Company, 1989.