Experiments in Program Synthesis with Grammatical Evolution: A Focus on Integer Sorting

Michael O'Neill, Miguel Nicolau, Alexandros Agapitos
UCD Complex & Adaptive Systems Laboratory
Natural Computing Research & Applications Group
University College Dublin
Ireland
Email: m.oneill@ucd.ie, miguel.nicolau@ucd.ie, alexagapitos@gmail.com

***Abstract*—We present the results of a series of investigations where we apply a form of grammar-based genetic programming to the problem of program synthesis in an attempt to evolve an Integer Sorting algorithm. The results confirm earlier research in the field on the difficulty of the problem given a primitive set of functions and terminals. The inclusion of a `swap(i,j)` function in combination with a nested `for` loop in the grammar enabled a successful solution to be found in every run. We suggest some future research directions to overcome the challenge of evolving sorting algorithms from primitive functions and terminals.**

I. Introduction

The application of GP to program synthesis has been under-explored by the community [1], and remains a significant open issue with many unanswered questions to be addressed [2]. This paper addresses this important research gap by applying a grammar-based form of Genetic Programming, Grammatical Evolution [3], [4], to the problem of synthesising a program to solve an integer sorting problem.

The remainder of the paper contains a summary of the relevant literature and background on program synthesis with GP in Section II. A description of the experimental setup and results in Sections IV and V respectively, and finally closing with Conclusions and possible directions for Future Work in Section VI.

II. Sorting Program Synthesis with GP

A literature review on sorting algorithm evolution revealed a limited repertoire of attempts in this problem domain. Most evolved algorithms were limited in the class of $O(n^2)$ such that bubble sort and insertion sort being evolved. Notable exception is the work of Agapitos et al. [5] that managed to evolve an $O(nlog(n))$ recursive sorting algorithm.

Kinnear [6], [7] evolved general iterative sorting algorithms, mainly of bubble sort's simplicity. He investigated the relative probability of success and the difficulty resulting from varying the primitive terminal and non-terminal elements. The primitive alphabet contained elements that could result in an exchange-oriented sorting strategy. Two primitive elements were used: `len` stored the length of the sequence to be sorted and `index` used as an iterator variable within the iteration control structure. Primitive functions were defined for swapping adjacent sequence elements, comparing elements on specified indices, comparing element values, incrementing and decrementing arithmetic variables. Control functions contained conditionals and a bounded iteration construct that extended the work reported in [8]. The fitness function followed the adjusted and normalized fitness discussed in [8] and was based on the number of *inversions*, a measure of sequence disorder [9]. This disorder was calculated before and after the evaluation of the evolved algorithm and the fitness was based on the post-execution remaining disorder. The fitness measure added a *linear parsimony function* to discourage the individuals' increasing size as well as a *disorder penalty*, in the case where the remaining disorder was greater than the initial disorder. During experimentation, Kinnear discovered a relation between program size and generality; adding inverse size to the fitness measure along with the quantification of its proximity to the target program, not only results in more parsimonious solutions, but also improves their generalization to unseen data. The sorting problem has then been used as a test-bed to evaluate several EA's control parameters and variation operators.

O'Reilly and Oppacher [10], [11] also investigated ways of evolving iterative sorting algorithms. Their first attempt [10] failed to produce a 100% correct individual. The primitive constructs and fitness functions used were different than those used in Kinnear's experiments. Specifically, primitive functions and control structures included decrementing a variable, accessing indexed sequence elements, swapping adjacent sequence elements, bounded looping, and conditional. The fitness function counted the number of out-of-place elements. Their second attempt [11] yielded a successful outcome. It considered different primitive constructs and two fitness functions; the first was used in [10] and the second was based on *permutation order* [9], the count of each element of the sequence of the smaller elements that follow it. The primitive setup used the same bounded iteration and element swapping constructs, and added two functions: `First-Wrong` and `Next-Lowest` that return the index of the first element that is out of order and the index of the smallest element after a particular indexed position, respectively.

Abbott [12], [13] used an Object Oriented Genetic Programming system to generate an insertion sort of quadratic complexity. He defined a `List` class of `Integer` object as a subclass of *Java*'s `ArrayList` class. His system operated

in a bigger program space by providing all methods declared in `ArrayList` as primitives for constructing hypotheses. Besides the standard `API` methods, two additional methods were declared: `iterate()` and `insertAsc()`. The former dictates the bounded iteration behavior, while the latter inserts its argument before the first element of the `List` that is greater than or equal to the argument. Using this alphabet, it was quite straightforward for an insertion sort to emerge. Although the set of primitive functions is sufficiently capable of expressing both an exchange- and insertion-oriented sorting recipe, Abbott does not report whether attempts were made to generate a sort program of bubble sort's structure.

Another attempt to the evolution of sorting algorithms is presented in the work of Spector *et al* [14] with their PushGP system. They used primitives along the lines of earlier investigations: swapping and comparing indexed elements, getting the list length, accessing list elements. The `Push3` programming language offers a variety of explicit iteration instructions but also allows for the evolution of novel control manipulation structures. They evolved an $O(n^2)$ general sorting algorithm and suggested an efficiency component addition to the fitness function as a precursor to the evolution of ingenious $O(n \times log(n))$ algorithms, though they reported no experiments towards that direction.

The most recent attempts to the evolution of recursive sorting algorithms are presented in the the work of Agapitos et al. [5], [15]. In [5] they studied the effects of language primitives and fitness functions on the success of the evolutionary process. For language primitives, these were the methods of a simple list processing package, plus the higher-order function filter. Five different fitness functions based on sequence disorder were evaluated, and the one by the name of Mean Sorted Position distance was able to evolve a sorting algorithm whose time complexity was measured experimentally in terms of the number of method invocations made, and was best approximated as $O(nlog(n))$.

In [15] they investigated the evolution of modular recursive sorting algorithms. They reported the computational effort required to evolve sorting solutions and provided a comparison between crossover and mutation variation operators, and also undirected random search. They found that the evolutionary algorithm outperformed undirected random search, and that mutation performed better than crossover. Modular sorting algorithms of insertion-sort type and complexity ($O(n^2)$) were evolved.

III. The Integer Sorting Problem Instance

In this study we explore the evolution of Python code to sort a list of integers from smallest to largest. The training cases used in this study are the following:

1) x = [1,0,2,3,4,5,6,7,8,9]
2) x = [9,8,7,6,5,4,3,2,1,0,11,10]
3) x = [1,0,3,2,5,4,7,6,9,8,14,11,10,12,13]
4) x = [1,0,9,8,7,6,5,4,3,2,11,10,19,18,17,16,15,14,13,12]
5) x = [0,7,2,3,5,4,6,1]

Note that we use unsorted input lists of variable sizes (length 10,12, 15, 20 and 8 respectively) and use integer values drawn from the range of zero to the input list length. The output list is initialised to be the same as the input list for each training case. To calculate the fitness of the evolved algorithms, for each training case we count the number of in-order pairs in the output list and divide this by the number of possible pairs in the list. Fitness is then the sum of these values subtracted from the number of test cases (we are minimising fitness). At the end of the paper we provide Fig. 12, which lists the Python code used to execute and calculate the fitness of the evolved programs.

We explore the use of different grammars starting from a simple primitive set of functions and terminals. Fig. 1 lists the first grammar examined which encodes a sequence of one or more `for` loops. The `for` loops are limited to iterate over the length of the input list, and therefore avoiding non-terminating programs. For those more familiar with languages like C, in a "non-Python" manner we explicitly declare and use a loop counter variable `i`. The set of language primitives is restricted to $\{$`i`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `for`, `if`, `+`, `-`, $>$, $<\}$ and the input list `x` and output list `guess`. Statements within a `for` loop are restricted to be either assignments or conditionals, where an assignment statements sets a value of the output list, and conditionals allow the comparison of values of the input and output lists to themselves or each other.

In the initial experiments a second grammar is employed (see Fig. 2) which is the same as the sequence of loops grammar in Fig. 1 except that it allows the generation of nested loops. The set of primitives is extended to include a loop counter variable (`j`) for the nested loop.

IV. Experimental Setup

We wish to determine if Grammatical Evolution can evolve solutions to the integer sorting problem as defined in Section III using either the sequence of loops or nested loops grammars (Fig.s 1 & 2).

One hundred independent runs were performed for each grammar using a population size of 500 for 100 generations. The same set of pseudo-random number generator seeds were employed for each setup. The following is a list of the other evolutionary parameters and their settings: one-point crossover probability of 0.9 with crossover forced to occur in expressed regions of the chromosome, an integer codon mutation rate of 0.1, tournament selection with a tournament size population ratio of 0.01, generational replacement with elitism (using population ratio of 0.1 elites), no wrapping, sensible initialisation (derivation tree minimum depth of 9 and maximum depth of 15 and adding a tail to the chromosome of 0.5 times its length).

V. Results

The left-hand side of Fig. 3 compares the mean best fitness averaged over one hundred runs for the Loops and Nested Loops grammars. A t-test at significance level 0.05 shows no

```
<code> ::= <for>
        | <for><code>

<for> ::= "\n"i=0"\n"
          for a in x :"\n\t"
               <for_a_in_x_line>"\n\t"
               i+=1

<for_a_in_x_line> ::= <for_a_in_x_setoutput>
                    | <for_a_in_x_cond>

<for_a_in_x_setoutput> ::=
guess\[<for_a_in_x_index>\] = <for_a_in_x_inputvar>"\n"

<for_a_in_x_index> ::= i
                     | ((i <biop> <const>)%TOTAL)

<for_a_in_x_inputvar> ::= x\[<for_a_in_x_index>\]

<for_a_in_x_outputvar> ::= guess\[<for_a_in_x_index>\]

<for_a_in_x_cond> ::=
"\n\t"if <for_a_in_x_expr><relop><for_a_in_x_expr> : <for_a_in_x_setoutput>

<for_a_in_x_expr> ::= <for_a_in_x_inputvar>
                    | <for_a_in_x_outputvar>
<biop> ::= + | -
<relop> ::= < | >
<const> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
i=0
for a in x :
     guess[i] = x[i]
     i+=1

i=0
for a in x :
     if guess[i+1]>x[i]: guess[i] = x[i]
     i+=1
```

Fig. 1. A grammar for a subset of the Python programming language which encodes a sequence of for loops (left) and an example individual generated by it (right). In the results of the experiments reported later this grammar is referred to as “Loops”.

```
<code> ::= <for>
        | <for><code>

<for> ::= "\n"i=0"\n"
             for a in x :"\n\t"
                  <for_a_in_x_line>"\n\t"
                  i+=1

<for_a_in_x_line> ::= <for_a_in_x_setoutput>
                    | <for_a_in_x_cond>
                    | <for_b_in_x>

<for_a_in_x_setoutput> ::=
guess\[<for_a_in_x_index>\] = <for_a_in_x_inputvar>"\n"

<for_a_in_x_index> ::= i
                     | ((i <biop> <const>)%TOTAL)

<for_a_in_x_inputvar> ::= x\[<for_a_in_x_index>\]

<for_a_in_x_outputvar> ::= guess\[<for_a_in_x_index>\]

<for_a_in_x_cond> ::=
"\n\t"if <for_a_in_x_expr><relop><for_a_in_x_expr> :
        <for_a_in_x_setoutput>

<for_a_in_x_expr> ::= <for_a_in_x_inputvar>
                    | <for_a_in_x_outputvar>

<biop> ::= + | -
<relop> ::= < | >
<const> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<for_b_in_x> ::= j=0"\n\t"
                 for b in x :"\n\t\t"
                      <for_b_in_x_line>"\n\t\t"
                      j+=1

<for_b_in_x_line> ::= <for_b_in_x_setoutput> | <for_b_in_x_cond>
<for_b_in_x_setoutput> ::=
guess\[<for_b_in_x_index>\] = <for_b_in_x_inputvar> "\n"

<for_b_in_x_index> ::= i
                     | ((i <biop> <const>)%TOTAL)
                     | j
                     | ((j <biop> <const>)%TOTAL)

<for_b_in_x_inputvar> ::= x\[<for_b_in_x_index>\]
<for_b_in_x_outputvar> ::= guess\[<for_b_in_x_index>\]
<for_b_in_x_cond> ::=
"\n\t\t"if <for_b_in_x_expr><relop><for_b_in_x_expr> :
        <for_b_in_x_setoutput>
<for_b_in_x_expr> ::= <for_b_in_x_inputvar>
                    | <for_b_in_x_outputvar>
```

```
i=0
for a in x :
     j=0
     for b in x :
          guess[i] = x[j+1]
          j+=1
     i+=1

i=0
for a in x :
     if guess[i+1]>x[i]: guess[i] = x[i]
     i+=1
```

Fig. 2. As per Fig. 1 this grammar encodes a sequence of for loops with the added possibility that nested loops can occur (left and right) and an example individual generated by it (bottom right). In the results of the experiments reported later this grammar is referred to as “Nested Loops”.

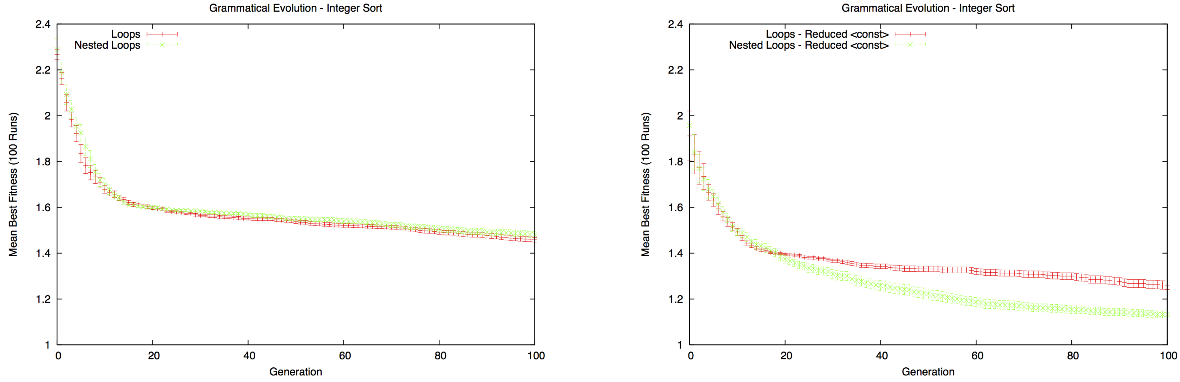


Fig. 3. Mean best fitness plots (with standard deviations) of the sequential loops and nested loops grammars (left) and their variants using a reduced set of constants. There is no statistical difference in performance in later generations of the different grammars with the standard set of constants, i.e., 0,..,9, while with a reduced set of constants, i.e, 0,1, the nested loops grammar is significantly different from the sequential loops setiup.

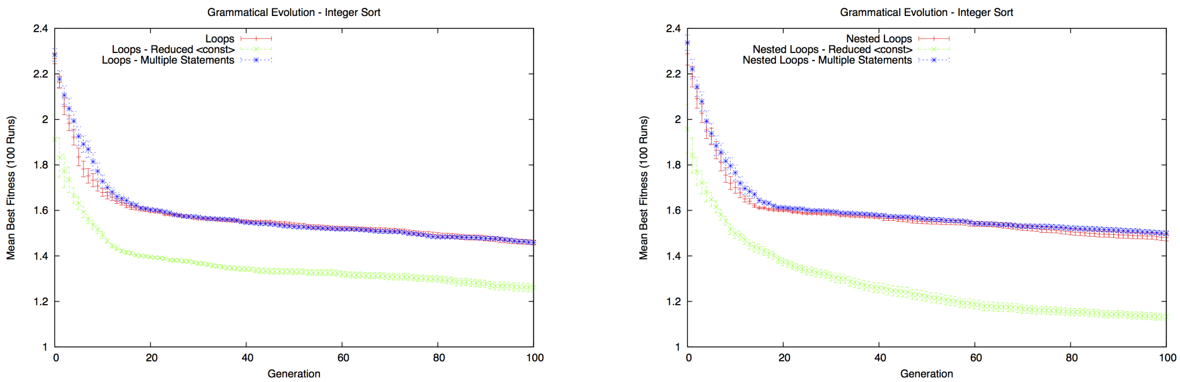


Fig. 4. A comparison of the mean best fitness (with standard deviations) of the initial two variations (reduced set of constants, and multiple statements in the for loops) of the sequential loops (left) and nested loops (right) grammars. The reduced set of constants setup clearly outperforms the other variations.

significant difference between the two populations of mean best values in the final generations.

A. Reduced set of constants

In a second set of experiments we reduced the search space encoded by the grammars by reducing the number of constants available from the integers zero to nine, to be restricted to just zero and one. Fig. 4 compares the mean best fitness averaged over one hundred runs for the Loops and Nested Loops grammars against the equivalents using a reduced set of constants. A t-test at significance level 0.05 shows there is a significant difference in the mean best fitness values, with the grammars adopting a reduced set of constants exhibiting performance gains.

Comparing the loops and nested loops grammars both using the reduced set of constants (see right-hand side of Fig. 3), we see a significant difference in performance for the Nested Loops grammar (t-test at level 0.05), however, a solution to the problem is not found.

B. Multiple statements within a for loop

In the third set of experiments we enable multiple statements to occur within the body of the for loops. The sequence of loops and nested loops grammars are extended as outlined in Fig. 5.

C. Increased population size

A fourth set of experiments were undertaken using an increased population size of 10,000 individuals over one hundred generations. A comparison to the original experiments (population size 500) is provided in Fig.'s 6 and 7. The larger population sizes result in a statistically significant difference in performance (t-test at 0.05), but again a solution to the problem is not found.

```
<for_a_in_x_line> ::= <for_a_in_x_setoutput>
                    | <for_a_in_x_cond>

is replaced with...

<for_a_in_x_line> ::= <for_a_in_x_setoutput>
                    | <for_a_in_x_cond>
                    | <for_a_in_x_line><for_a_in_x_line>
```

```
<for_b_in_x_line> ::= <for_b_in_x_setoutput>
                    | <for_b_in_x_cond>

is replaced with...

<for_b_in_x_line> ::= <for_b_in_x_setoutput>
                    | <for_b_in_x_cond>
                    | <for_b_in_x_line><for_b_in_x_line>
```

Fig. 5. The sequential loops grammar (Fig. 1 is modified as illustrated here (left) and the nested loops grammar (Fig. 2) is modified to allow multiple statements inside each for loop, using both the changes illustrated (left and right).

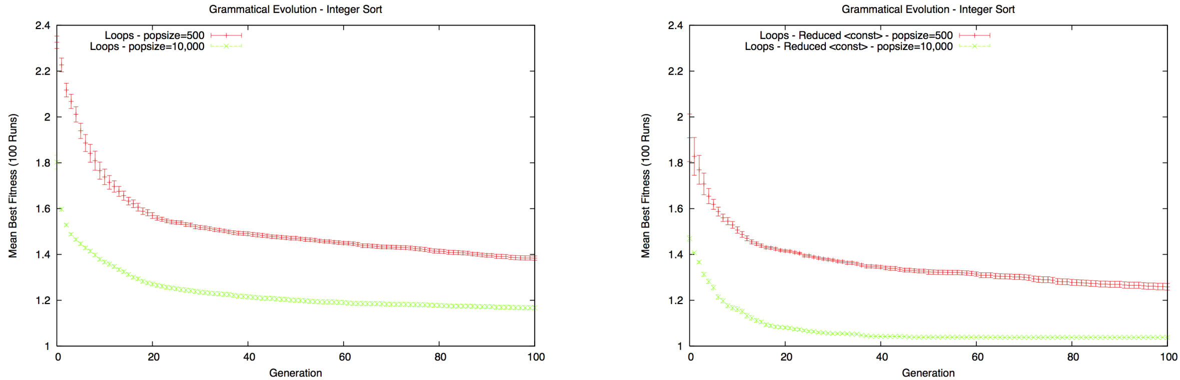


Fig. 6. Mean best fitness plots (with standard deviations) of the original population size (500) versus an increase to 10,000 using the sequential loops grammar. The larger population sizes produce significantly better results.

D. Restricted crossover sites

A fifth set of experiments were undertaken using a modified grammar where crossover sites are specified to restrict crossover events to exchanging `if` statement conditions. An extract from the modified loops grammar is detailed below, which utilises the special non-terminal symbol <GEXOMarker>.

```
if <GEXOMarker><for_a_in_x_expr><GEXOMarker>
   <relop>
   <GEXOMarker><for_a_in_x_expr><GEXOMarker> :

        <for_a_in_x_setoutput>
```

By specifying crossover sites in the grammar, crossover is only allowed to occur between codons responsible for generating condition expressions. Results are presented in Fig. 8. We do not observe performance gains over the nested loops grammars.

E. Adding swap(i,j)

Earlier research using different forms of Genetic Programming on a similar problem, found that the addition of higher-level primitives can enable a solution to a sorting problem to be found. To this end, in a final experiment, to each grammar we added a swap function which switches the values of the output list (guess) at indices i, and j. All runs of the Nested Loop grammar now successfully solve the problem with the addition of swap(i,j), however, the sequence of loops grammar still fails to solve the problem.

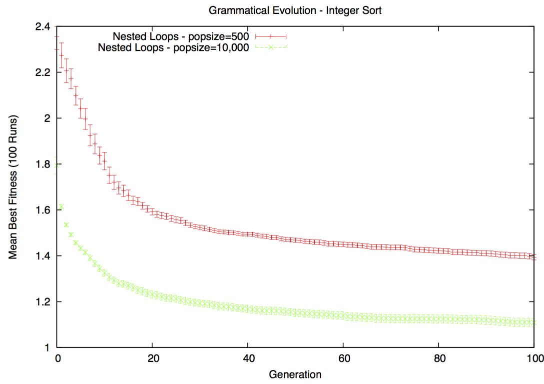
F. Examples of Evolved Programs

Examples of evolved sorting programs are presented in Fig. 10 and Fig. 11.

VI. Conclusions & Future Work

The results presented here demonstrate the difficulty a Genetic Programming algorithm has in solving an integer sorting problem using a primitive set of functions and terminals. Following examination of a number of variants of the grammars and different population sizes we discovered that the inclusion of a swap function with nested loops successfully solved the problem easily using standard population sizes for toy GP problems (i.e., 500).

These results using Grammatical Evolution, confirm the findings of earlier studies on variations of the sorting problem using other forms of Genetic Programming, which found it challenging to evolve a general sort algorithm using low-level primitives. This suggests further research is required on this benchmark problem to determine what is missing from GP in order to allow it to successfully evolve solutions from low-level primitives. One approach we will explore, and are optimistic on its success is to employ the use of semantic methods. For example, search operators which compare the semantics of the parents may be beneficial in a more effective exploration of the space of programs allowing us to manage semantic locality and diversity [16].



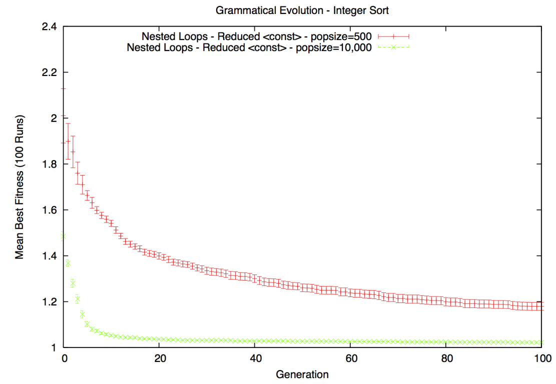
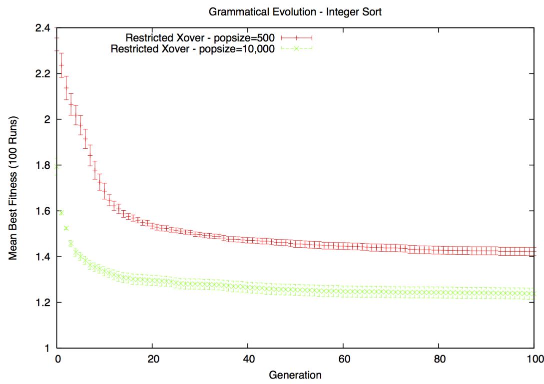


Fig. 7. Mean best fitness plots (with standard deviations) of the original population size (500) versus an increase to 10,000 using the nested loops grammar. The larger population sizes produce significantly better results.



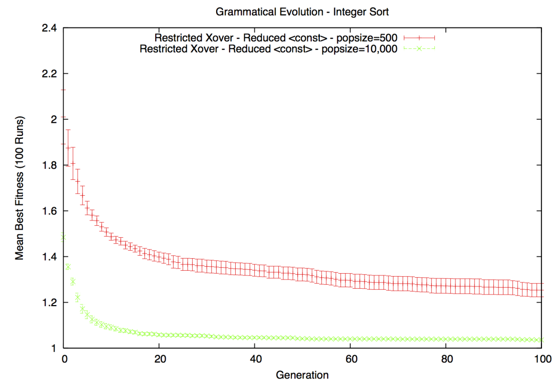
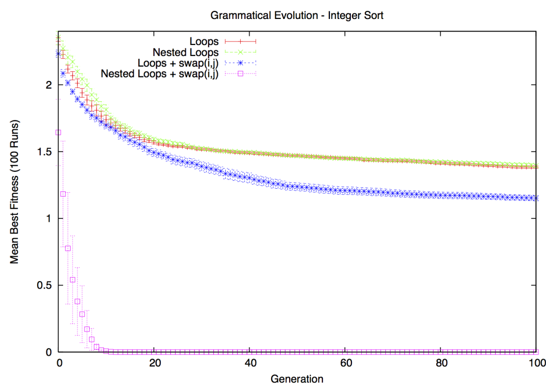
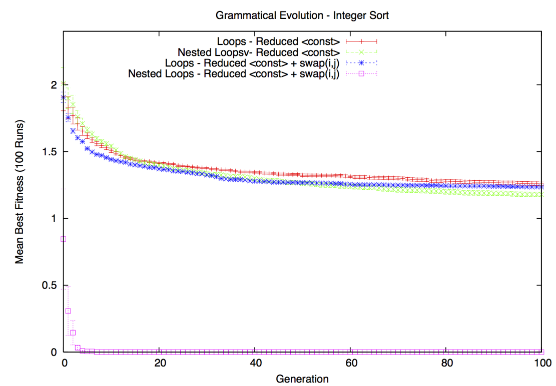


Fig. 8. Mean best fitness plots (with standard deviations) of the restricted crossover grammars with the small and large population (left) versus the restricted crossover grammar combined with a reduced set of constants (right). The variant which uses a reduced set of constants exhibits the best performance. As per Fig. 7 the larger population sizes produce significantly better results.



(a) Loops and Nested Loops grammars with the addition of swap(i,j).



(b) Reduced Constants versions of the Loops & Nested Loops grammars.

Fig. 9. Adding swap(i,j) to the set of primitives allows the Nested Loop grammars to solve the sorting problem in each of the 100 runs.

```
i=0
for a in x :
        j=0
        for b in x :

                if guess[j]>guess[i] : swap(((j - 0)%TOTAL),i)

                j+=1
        i+=1
```

Fig. 10. A successfully evolved sort algorithm using the Nested Loops grammar including the swap(i,j) function. This solution was found in generation 10 (population size 500).

```
i=0
for a in x :
        j=0
        for b in x :

                if guess[j]>guess[((i + 0)%TOTAL)] : swap(i,j)

                j+=1
        i+=1
```

Fig. 11. A successfully evolved sort algorithm using the Nested Loops grammar with the reduced set of constants including the swap(i,j) function. This solution was also found in generation 10 (population size 500).

REFERENCES

[1] D. R. White, J. McDermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaskowski, 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.

[2] M. O'Neill, L. Vanneschi, S. Gustafson, and W. Banzhaf, "Open issues in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3/4, pp. 339–363, Sep. 2010, tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines.

[3] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, ser. Genetic programming. Kluwer Academic Publishers, 2003. [Online]. Available: http://www.wkap.nl/prod/b/1-4020-7444-1

[4] I. Dempsey, M. O'Neill, and A. Brabazon, *Foundations in Grammatical Evolution for Dynamic Environments*, ser. Studies in Computational Intelligence. Springer, 2009. [Online]. Available: http://www.springer.com/engineering/book/978-3-642-00313-4

[5] A. Agapitos and S. M. Lucas, "Evolving efficient recursive sorting algorithms," in *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*. Vancouver: IEEE Press, 6-21 Jul. 2006, pp. 9227–9234. [Online]. Available: http://privatewww.essex.ac.uk/~aagapi/papers/AgapitosLucasEvolvingSort.pdf

[6] K. E. Kinnear, Jr., "Generality and difficulty in genetic programming: Evolving a sort," in *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, S. Forrest, Ed. University of Illinois at Urbana-Champaign: Morgan Kaufmann, 17-21 Jul. 1993, pp. 287–294.

[7] ——, "Evolving a sort: Lessons in genetic programming," in *Proceedings of the 1993 International Conference on Neural Networks*, vol. 2. San Francisco, USA: IEEE Press, 28 Mar.-1 Apr. 1993, pp. 881–888.

[8] J. Koza, *Genetic Programming: on the programming of computers by means of natural selection*. Cambridge, MA: MIT Press, (1992).

[9] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, USA: Addison Wesley Longman Publishing, 1998.

[10] U.-M. O'Reilly and F. Oppacher, "An experimental perspective on genetic programming," in *Parallel Problem Solving from Nature 2*, R. Manner and B. Manderick, Eds. Brussels, Belgium: Elsevier Science, Sep. 28 - 30 1992, pp. 331–340. [Online]. Available: http://www.ai.mit.edu/people/unamay/papers//ppsn92.ps

[11] ——, "A comparative analysis of GP," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr., Eds. Cambridge, MA, USA: MIT Press, 1996, ch. 2, pp. 23–44.

[12] R. Abbott, J. Guo, and B. Parviz, "Guided genetic programming," in *The 2003 International Conference on Machine Learning; Models, Technologies and Applications (MLMTA'03)*. las Vegas: CSREA Press, 23-26 Jun. 2003.

[13] R. Abbott, B. Parviz, and C. Sun, "Genetic programming reconsidered," in *Proceedings of the International Conference on Artificial Intelligence, IC-AI '04, Volume 2 & Proceedings of the International Conference on Machine Learning; Models, Technologies & Applications, MLMTA '04*, H. R. Arabnia and Y. Mun, Eds., vol. 2. Las Vegas, Nevada, USA: CSREA Press, Jun. 21-24 2004, pp. 1113–1116. [Online]. Available: http://abbott.calstatela.edu/PapersAndTalks/GeneticProgrammingReconsidered.pdf

[14] L. Spector, J. Klein, and M. Keijzer, "The push3 execution stack and the evolution of control," in *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, New York, NY, USA, 2005, pp. 1689–1696.

[15] A. Agapitos and S. M. Lucas, "Evolving modular recursive sorting algorithms," in *Proceedings of the 10th European Conference on Genetic Programming*, ser. Lecture Notes in Computer Science, M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, Eds., vol. 4445. Valencia, Spain: Springer, 11 - 13 Apr. 2007, pp. 301–310.

[16] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and D. N. Phong, "On the roles of semantic locality of crossover in genetic programming," *Information Sciences*, vol. 235, pp. 195–213, 20 Jun. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025513001175

```
import math
import sys

TOTAL = 10                                          #global variable containg the input list length
guess = [(TOTAL-ii-1) for ii in range(0, TOTAL)]    #output list
x = [0.0 for ii in range(0, TOTAL)]                 #input list (to be sorted)

def swap(r,s):                                      #swap(i,j) function
        global guess                                #not so elegant implementation
        t0 = guess[r]
        t1 = guess[s]
        guess[r] = t1
        guess[s] = t0

def loadTestCase1():                                #test case 1
        global TOTAL
        global x
        x = [1,0,2,3,4,5,6,7,8,9]
        TOTAL = len(x)

def loadTestCase2():                                #test case 2
        global TOTAL
        global x
        x = [9,8,7,6,5,4,3,2,1,0]+[11,10]
        TOTAL = len(x)

def loadTestCase3():                                #test case 3
        global TOTAL
        global x
        x = [1,0,3,2,5,4,7,6,9,8,14,11,10,12,13]
        TOTAL = len(x)

def loadTestCase4():                                #test case 4
        global TOTAL
        global x
        x = [1,0,9,8,7,6,5,4,3,2,11,10,19,18,17,16,15,14,13,12]
        TOTAL = len(x)

def loadTestCase5():                                #test case 5
        global TOTAL
        global x
        x = [0,7,2,3,5,4,6,1]
        TOTAL = len(x)

def calculateSortDifferenceInOrderPairs(cmd):       #count the number of ``in-order'' pairs
        global guess
        guess = [x[ii] for ii in range(0, TOTAL)]   #initialise the output list to the input list values
        exec cmd
        inOrderPairCount = 0.0
        pairs = TOTAL-1
        for ii in range(1, TOTAL):
                if (guess[ii] > guess[ii-1]):
                        inOrderPairCount += 1
        return (inOrderPairCount / pairs)
```

Fig. 12. The Python code used to execute the evolved code and calculate fitness.