Effects of Population, Generation and Test Case Count on Grammatical Genetic Programming for Integer Lists

Hakan Ayral*, Songül Albayrak*
Yildiz Technical University, Computer Engineering Department, Istanbul, Turkey.

* Corresponding author. email: hayral@gmail.com, songul@ce.yildiz.edu.tr
Manuscript submitted February 5, 2017; accepted May 3, 2017.
doi: 10.17706/jsw.12.6.483-492

Abstract: This paper investigates how grammatical genetic programming performs for evolving simple integer list manipulation functions. We propose three sub-problems which are related to, or component of integer sorting problem as defined by genetic programming literature. We further investigate the effects of modifying evolutionary parameters, such as the number of generations allowed, number of populations, and number of test cases, on the number and distribution of successful solutions. Finally, we propose an AST based dead-code removal for the intron induced non-functional codes on evolved individuals.

Key words: BNF grammar, evolutionary computing, formal language, grammatical genetic programming.

1. Introduction

Genetic programming (GP) is a population based evolutionary search technique similar to genetic algorithms, where evolved objects are "programs" of varying forms, like a simple mathematical expression, a complete decision tree, or the source code of a complete program. Fitness functions are commonly lists of test cases in form of (input, output) pairs, along with a distance function to serve as a metric between the computed output and expected output; these two are the only apriori information available to guide the search in the space of valid programs belonging to language defined by given grammar.

As expected the fitness landscape of such a space is not smooth. For GP and especially grammatical GP, fitness landscapes are not differentiable to exploit gradients. They also have very poor locality and structure. It is hard to define closeness in the infinite space of programs belonging to a language and conserving some attributes so that locality can be exploited to guide the search towards the solution, at the same time.

Grammatical GP is a form of GP where the search space is reduced to the space of syntactically valid programs according to a given a formal grammar. Excluding invalid programs dramatically reduce the search space, which is still infinite but has a lower dimensionality. On the other hand a syntactically valid program is not necessarily semantically meaningful too; as a matter of fact most of them are not.

In this paper we propose three algorithmic problems on manipulating integer lists as sub components of *integer sort problem*; and evolve solutions for them using grammatical GP. We define a high level algorithmic problem as those which involve loops and internal state variables, as opposed to just evolving expressions or decision trees. According to survey [1], almost 90% of the papers published in EuroGP and GECCO GP between years 2009-2012 belonged to Symbolic regression, Classification or Boolean functions (i.e. parity, multiplexers) domains, which aren't of algorithmic nature on this sense.

The difficulty of evolving a high level algorithm like sort, using a grammar consisting of low level

constructs only, has been identified on [2], where a successful integer sort algorithm can be evolved only when the high level swap operation is introduced to the grammar, even though the grammar used is already highly customized towards sorting. Evolving a general integer sort algorithm that run correctly on arbitrarily sized lists, using only low level syntactic constructs is still an open problem.

2. Search Space, Grammars and Difficulty

2.1. Generality of a Grammar

Let us define what we mean by the generality of a grammar. High level programming languages contain constructs such as assignments, loops and conditionals as building blocks; they are declared as an ordered list of statements and can be grouped in code blocks which may contain other code blocks. Furthermore assignments are made towards variables which hold the algorithm state, with expressions producing new values by combining other values and variables through some operators.

We use the term "unconstrained grammar" to describe the general case where there are no bounds on the number of statements in a code block, on the number of code blocks which can be nested, on the number of operands and operators forming an expression and on the number of variables to hold the program state.

Almost all programming languages have unconstrained general grammars. Some constrained processing hardware impose limits on the number of variables based on available registers, on number of statements based on instruction pointer width, and on number of nested function calls based on hardware stack size; also heterogeneous computing platforms (i.e. OpenCL, CUDA) have limits on number of local variables, availability of array indexing and total number of statements based on underlying hardware, but these are enforced at compiler level as compilation errors, instead of grammar level as syntax errors.

Conversely we define a grammar as constrained, when these limits are incorporated explicitly on the BNF definition of the grammar. The purpose of using a constrained grammar in GP is to prune the search space by incorporating some apriori constraints to the language, based on how the answer being searched is supposed to look like. This specialization of grammar biases the language towards the answer.

2.2. Problem Difficulty in Genetic Programming

Difficulty in GP has been a popular subject of study [3]–[7], and difficulty of evolving an integer sort code has been identified relatively early [8]. We have replicated the results in [2] where it is stated that if a high level construct like a built in swap function is not provided as part of the language defined by grammar the evolution does not converge to a solution for the case of *integer sort* problem. To the best of authors' knowledge [2] dated 2014 is the most recent work on literature dealing with the *integer sort* problem.

2.3. Unbounded Expansions on General Grammars

A grammar is defined as a set of production rules which maps tokens to strings of tokens and literal characters; this in turn defines a language which is a set of programs. In the case of an unconstrained grammar the language defined has infinitely many elements, as you can write infinitely many different programs which are valid in the defined language. Then the natural question that arises is about identifying the dimensionality of this set.

When expanding the production tree having the start token of a grammar as the root node, leaf nodes with tokens corresponding to non-terminal production rules expand to child nodes consisting of literal strings and other tokens. If a non-terminal token contains itself in one of its expansion rules, or a loop can be constructed by cycling through a set of non-terminals referring each other in their expansions, then it means that some infinite expansions are also part of the language along with the programs they represent.

Such infinite trees do not represent any program, as some branches never reach terminal tokens; but in the same time each different ways to construct them represent a different axis corresponding to a

dimension of the space of programs in the defined language. A first example comes from the allowed number of statements in a code block; on an unconstrained grammar this can be expressed with a recursive expansion rule, such as "*<statements> ::= <statement> | <statement><statements>*". The *statements* token has 50% chances to get expanded to itself along with a new statement token. Even though the expected value of the number of statement tokens generated by this rule is $\sum_{n=1}^{\infty} n\frac{1}{2^n} = 2$, the distribution of it has a long tail; therefore arbitrarily large number of consecutive statements can be observed, but with exponentially decaying probabilities. A further complication is that what we have isn't a real random process, and the source of randomness to choose from possible expansion rules is provided by the finite genome of an individual, which is a random list of integers with length in the order of hundreds. When a request for next random value exhaust the values on the genotype of individual, the list is recycled and used again from the start; this is the most common solution employed, called wrap around in the literature, but it generate a periodic correlation on the stream of values read. When hundreds of individuals over hundreds of generations are considered, the periodicity due to size of the genome, and periodicities that arise due to cyclic arrangements of non-terminal expansion rules, may coincide on a common period and form an infinite expansion loop without ever reaching a terminal rule. This is a known phenomenon; some alternatives to wrap-around reuse of genotype has been studied and proposed in [9] and [10].

The number of statements growing unbounded is one of the dimensions a general grammar can go expanding indefinitely; the other two dimensions are the number of nested code blocks and expression length, as shown on Fig.1. In the ideal case of genotypes with no periodicity (i.e. infinitely long random genotype) expansion trees are expected to stop eventually before diverging to infinity, as the probability of having the same non-terminal expansion sequence decays exponentially at each step; yet it is a very large space with no hope for success while employing exhaustive search or random search.

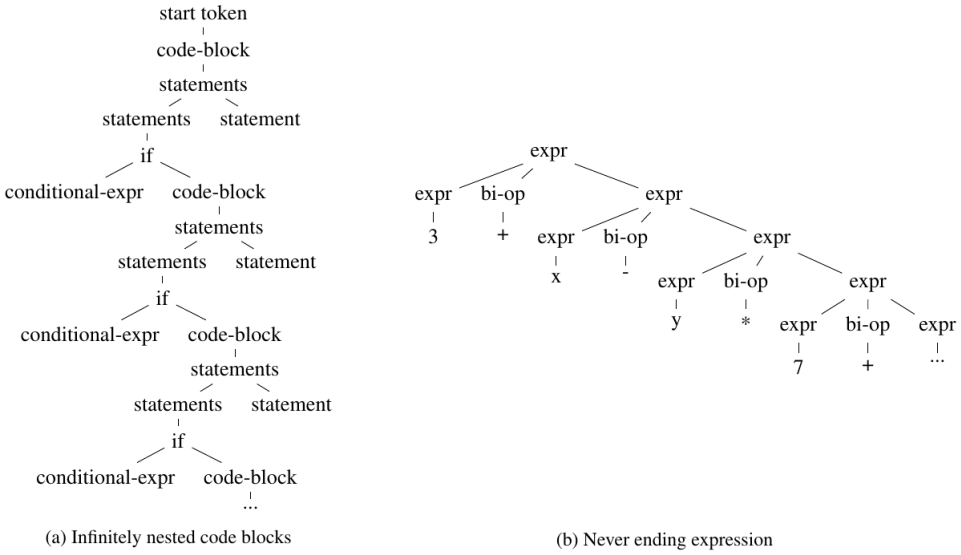

Fig. 1. Two unbounded expansion trees.

2.4. Expansion Order

Another issue with applying expansion rules is how to select the next non-terminal to expand when there is more than one. Obviously a depth first selection will always expand the left-most non-terminal node. A breadth first traversal order is possible but tricky, as each expansion also modifies the tree. It can be implemented by tracking all non-terminal child nodes in a FIFO queue to impose an order, with new non-terminals added to the end of the queue, and the next non-terminal to expand is retrieved from the front; when the queue becomes empty it means that all leaf nodes of the tree are terminal nodes.

Independent of in which order we choose the next non-terminal leaf to expand, without some extra information about the rest of the tree or at least of the branch, it is impossible to limit the unbounded expansion as the grammar doesn't have that information.

2.5. Setting Hard Limits on Expansions

A way to provide depth information at expansion time is to introduce redundant copies of expansion rules to the BNF differentiated by their depth. Let's consider the expansion rule "*<expr> ::= <expr> <bi-op> <expr> | <int> | <var>*" with possible infinite expansions as illustrated no Fig.2(b) Its first production rule is both left-recursive and right-recursive. To exclude infinite expansions of this type from the language, and limit the maximum depth up to a constant K, we replace this rule with multiple depth annotated copies of the form "*<expr-N> ::= <expr-N+1> <bi-op> <expr-N+1> | <int> | <var>*" for each $N \in \{1,2,\dots,K-1\}$ and a last rule of form "*<expr-K> ::= <int> | <var>*" which omits the recursive part. This new set of *K* rules allows expressions consisting of an integer, a variable, or a combination of those with up to *K-1* binary operators.

The same annotation can be extended to bound the expansion depth of cyclical self-references involving multiple non-terminals as illustrated on Fig.2.

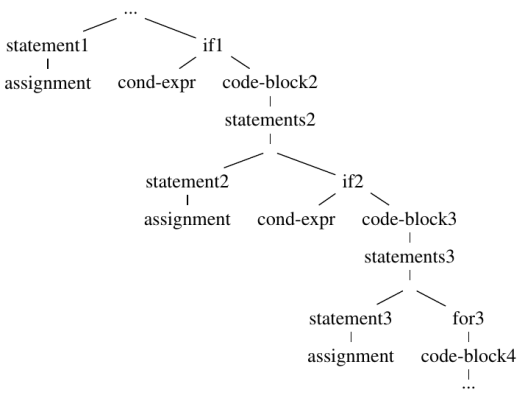

Fig. 2. Depth information by duplication of rules in BNF.

The information provided by this static bounding by duplication of rules with counters, is the current depth on the branch. This allows tracking expansion depth on a derivation branch involving multiple types of non-terminals, but it requires a consistent numbering across duplicates of all types of tokens involved, but still the information is branch local and doesn't provide information about the rest of the tree.

3. Proposed Problems

3.1. Min Problem

As an easier alternative to integer sort, we considered the *Min Problem* as a good candidate to evolve; which consists of finding the minimum value in a given list of integers. It can also be thought as a components of integer sort; if we compose the sort as "*for each position of the list, swap the value at that position with the minimum value at the rest of the list*" which is equivalent to insertion sort. We devised a static bounding grammar for the *Min Problem* which is a compromise between being a general enough grammar, and not expanding too much in terms of tree size. See Appendix for the proposed BNF grammar.

3.2. Search Problem

Search Problem consists of identifying whether a given search value is present in an integer list and return true or false accordingly. It is not a component of integer sort, but still an elementary function that can be evolved with GP. We implemented the *Search Problem* with a static bounding grammar. As the *Search Problem* is a question with a binary answer, it is interesting for its use of both integer and boolean values.

3.3. Search with Position Problem

The hardest problem we managed to evolve correct solution is *Search with Position* finding a function which searches for a value in an integer list, and returns its position if present, or return -1 if number is not present. See Appendix for the proposed BNF grammar to evolve solutions for this problem.

4. Experiment Results

4.1. Baseline Experiment

We used 50 populations for the evolution of *Search With Position* grammar, using a population size of 100 individuals. There were 40 test cases with half of them containing the searched integer; lists had length varying between 3 to 7. Same test cases have been used for all populations; crossover and mutation rate were both set to 0.7. We employed elitism and two way tournament selection on all experiments.

It can be seen on Fig.3 that 12 out of 50 populations managed to evolve a successful individual that can return the position of the searched value or -1 if it isn't in the list. Fig.3 consists of two fitness trace plots; one for successful populations (those who managed to evolve an individual with zero error before 500 generations) and another for failed populations. This separation of plots is purely to better portray the difference of distribution characteristics in successful and unsuccessful populations. Each trace line corresponds to a population, and values attained by a line are the fitness values of the best individual of that population at that generation. Therefore vertical jumps mean a new best individual has evolved in that population which has lower error than previous best; and horizontal segments mean the population couldn't evolve a new best individual, so the one from previous generation has been preserved by elitism.

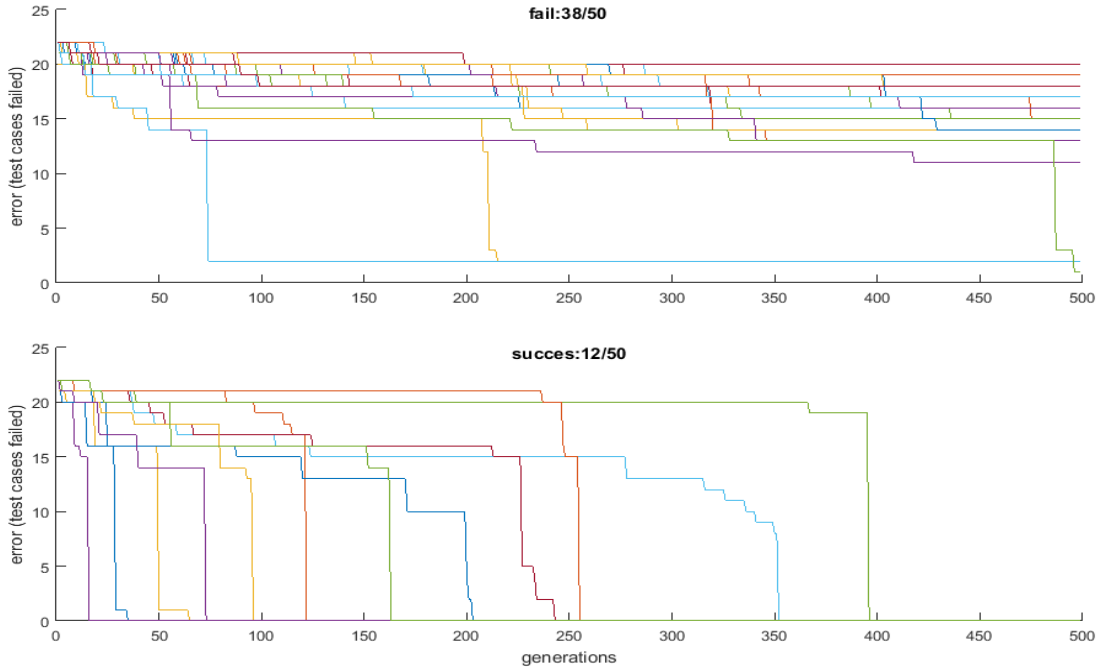

Fig. 3. Fitness traces of best individuals of 50 runs for 500 generations.

The average fitness of best individuals is only meaningful for the failed populations; successful populations leave the process when a zero error individual is evolved, therefore the number of populations remaining (to average over) is different for different points of horizontal axis, making the same average for successful populations hard to define. Fig.4 shows the mentioned average fitness of best individuals of failed populations; jagged traces can be seen to converge to a smoother curve upon averaging.

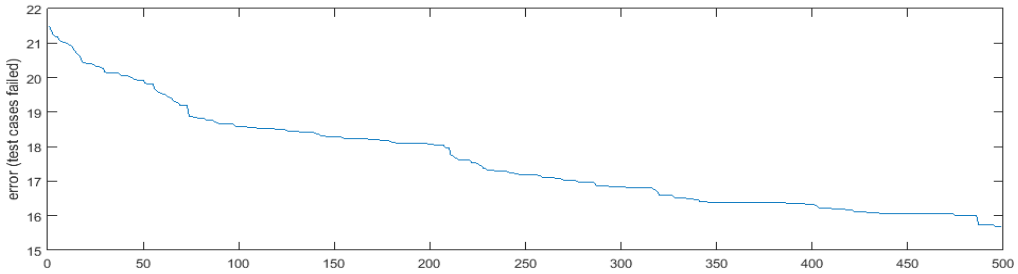

Fig. 4. Fitness average of failed populations' best individuals.

4.2. More Populations, Less Generations, Same Wall-Clock Time

The distribution of Fig.3 hints that the probability of a population to achieve zero error gets lower as the number of generations increase. Thus, we repeated the experiment with the same settings, except limiting the maximum number of generations fivefold (from 500 to 100). To obtain an equivalent computation time, we also increased the number of populations fivefold (from 50 to 250). Fig.5 shows the traces of the experiment, where 32 out of 250 populations managed to produce an error free individual. Although the success rate drops to 0.128 (32/250) from previous 0.24 (12/50), trying more independent populations for shorter times yielded 2.66 (32/12) times more error free individuals for the same computation time. Notice that the points where populations reach an error free individual are distributed more homogeneously across 100 generations, in contrast to previous experiment. Yet, the points at which zero error individuals are reached doesn't get more frequent at left edge of horizontal axis neither, to the contrary the first 15 generations see no solutions, in stark contrast to the generation range 15-100.

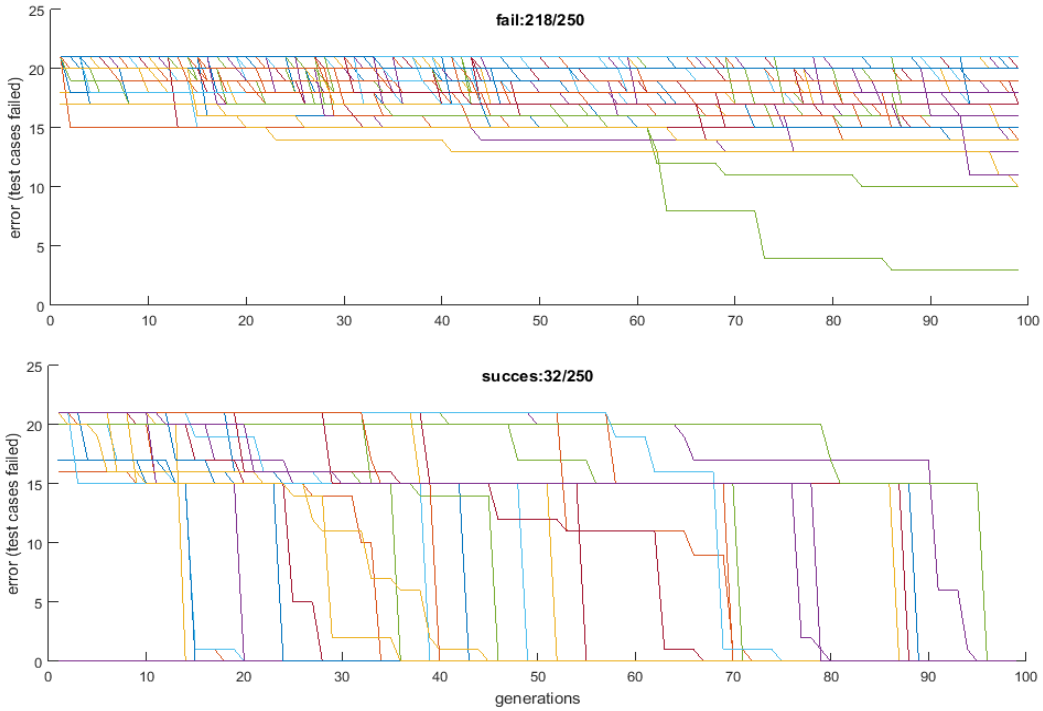

Fig. 5. Fitness trace of best individuals for 250 population over 50 generations.

4.3. Less Test Cases

We conducted a third set of experiments, with the same settings as the first one, but this time decreasing the number of test cases from 40 to 20, which halves the computation time in exchange for an increased risk

of false positives labeled as successful. Figures Fig.7 and Fig.8 show the traces of best individuals of populations, and the average of best individuals from unsuccessful populations. The success rate became 0.26 (13/50) with an increase of 2% success rate, which is statistically insignificant to attribute to false positives which may stem from a lack of test cases.

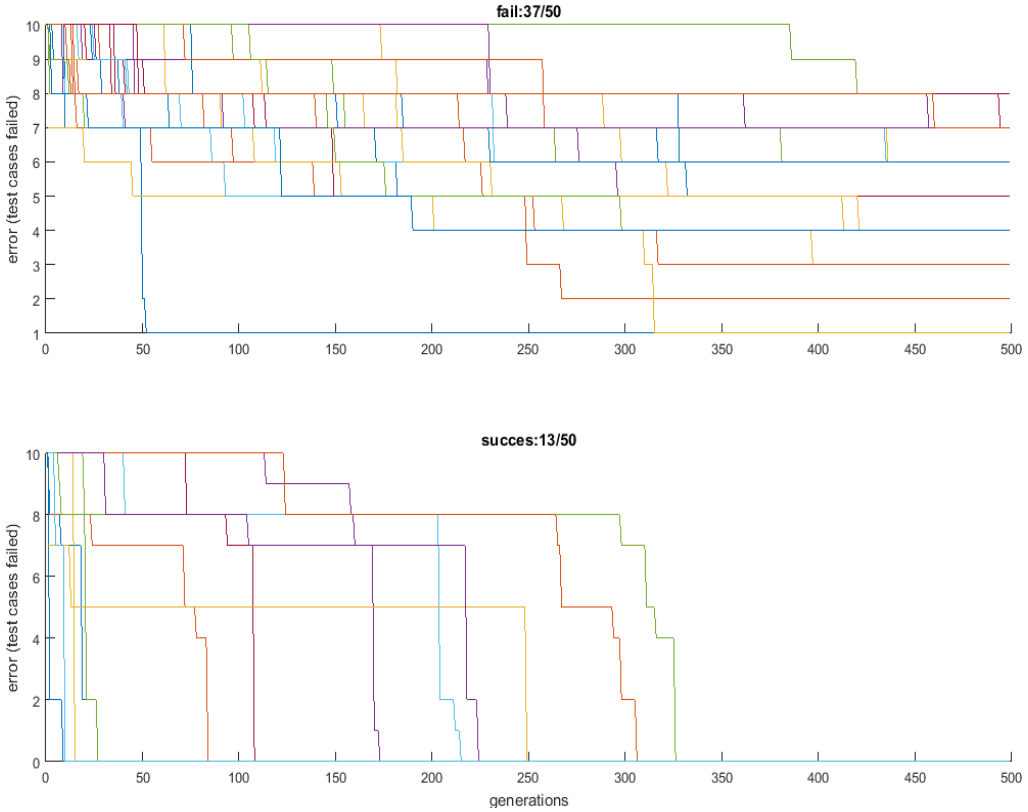

Fig. 7. Fitness trace of best individuals for 50 populations using half the test cases.

The traces for successful populations on figure Fig.7 shows that 38% (5/13) of solutions are concentrated in the narrow range of first 25 generations. It can be conjectured that the decreased number of test cases contributed to early success of evolution in contrast to previous two setups.

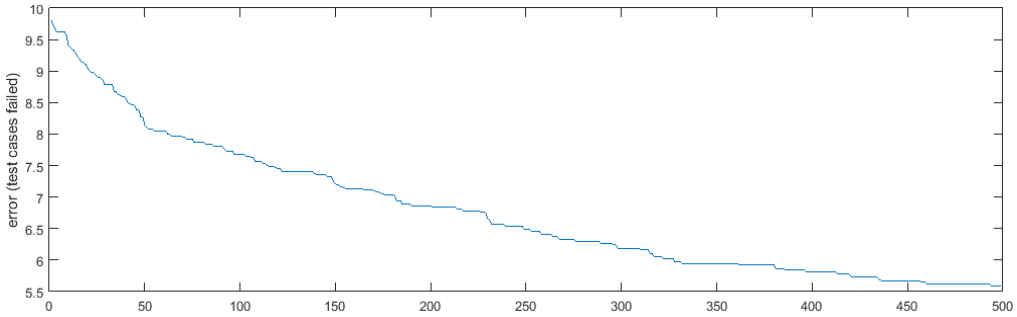

Fig. 8. Fitness average of populations' best individuals.

The rate of false positives as a function of number of test cases will be a parameter we will further investigate. As the number of test cases affect the computation time linearly (i.e. doubling the test cases doubles the computation time), it may be beneficial to use minimum indicative number of test cases. Total computation time can be minimized by allowing a higher false positive rate and thoroughly checking the individual solutions later, instead of thoroughly checking the whole population earlier.

Another parameter to investigate is the cumulative density of the probability distribution of having an

error free individual in terms of number of generations. The extreme case of allowing only one generation is equivalent to random search, as the populations don't get a chance to cooperate through crossover.

5. Automated Dead Code Elimination

Listing below shows one of the solutions evolved for *Search With Position* problem. It can be seen that the effect of many lines of the code gets overwritten by later code. Dead code is a common issue with GP [11], it is a byproduct of introns (non-coding parts of the genotype). For a detailed investigation of introns in grammatical GP see [12].

```
def createdFunc(INPUT,SEARCH):
        LENINPUT = len(INPUT)
        v = 0
        i = 0
        tmp = 0
        OUTPUT = -1
        if (-1) + SEARCH != tmp - v:
                if 1 > v:
                        i = 1 + v
                        tmp = 2 + OUTPUT
        if SEARCH == v - OUTPUT:
                if OUTPUT + SEARCH == 1:
                        i = SEARCH
                        OUTPUT = 2
                OUTPUT = SEARCH - OUTPUT
        i=0
        for v in INPUT :
                if SEARCH == v:
                        OUTPUT = SEARCH
                        OUTPUT = i
                i+=1
        return OUTPUT
```

```
def createdFunc(INPUT, SEARCH):
    OUTPUT = -1
    i = 0
    for v in INPUT:
        if (SEARCH == v):
            OUTPUT = SEARCH
            OUTPUT = i
        i += 1
    return OUTPUT
```

Listing. 1. A solution for *Search With Position* problem (left) with dead-code (right) dead-code removed

Dead code elimination has no effect on the end result of how the evolved code works; it just provides means to see more clearly the actual part of code which produces the desired result. Trying to remove each line of the evolved code would break the syntax on lines that defines a new code block; to prevent this we parse the evolved code back to AST, disable each immediate child of root node one by one to build candidate trees, and render them back to source code to be run against test cases. All candidate source codes are syntactically valid Python source codes, but functionality isn't necessarily preserved. We run each candidate against same test cases, if the omission of a node still produces fully functional code we incorporate that omission to original tree, and call the dead code elimination routine recursively for next depth on this new tree to search for deeper nodes that can be eliminated.

6. Conclusion

In this paper we propose three integer list manipulation problems based on their relation to *integer sort* problem, and evolve solutions for those using grammatical GP. *Min Problem* requires finding a function which returns the smallest value in a list. *Search Problem* requires finding a binary function returning True only if a given value is present in a given list, and False otherwise. Finally *Search with Position Problem* requires a function which returns the position of a given value in a given list, or -1 if the value is not present. We have given grammars for problems and successfully evolved correct code for all three of them. Furthermore we investigated the effect of modifying the number of generations, populations, and test cases allowed, on the number and distribution of successful solutions. Finally we present a dead-code elimination algorithm to automatically remove nonfunctional code introduced by introns.

Appendix (BNF Grammars)

```
<expr>        ::= <expr2> <bi-op> <expr2> | <expr2>
<expr2>       ::= <int> | <var>
<var>         ::= tmp | i | OUTPUT | v | INPUT[<var> % LENINPUT]
<var-write>   ::= tmp | i | OUTPUT | v
<bi-op>       ::= + | -
<int>         ::= 1 | 2
<statement>   ::= <assignement> | <if> | <loop>
<statement2>  ::= <assignement> | <if2>
<statement3>  ::= <assignement>
<loop>        ::= i=0<nl>for v in INPUT :<nl><c-block2><nl>\ti+=1
<if>          ::= if <cond-expr>: <nl><c-block2>
<if2>         ::= if <cond-expr>: <nl><c-block3>
<cond-expr>   ::= <expr> <comp-op> <expr>
<comp-op>     ::= #lesser# | #greater# | == | !=
<assignement> ::= <var-write> = <expr>
<c-block>     ::= <statements>
<c-block2>    ::= <statements2>
<c-block3>    ::= <statements3>
<statements>  ::= <statement><nl>
                | <statement><nl><statement><nl>
                | <statement><nl><statement><nl><statement><nl>
<statements2> ::= <statement2><nl>
                | <statement2><nl><statement2><nl>
                | <statement2><nl><statement2><nl><statement2><nl>
```

```
<expr>        ::= <expr2> <bi-op> <expr2> | <expr2>
<expr2>       ::= <int> | <var>
<var>         ::= tmp | i | v | OUTPUT | INPUT[<var> % LENINPUT] | SEARCH
<var-write>   ::= tmp | i | v | OUTPUT
<bi-op>       ::= + | -
<int>         ::= 1 | 2 | (-1)
<statement>   ::= <assignement> | <if> | <loop>
<statement2>  ::= <assignement> | <if2>
<statement3>  ::= <assignement>
<loop>        ::= i=0<nl>for v in INPUT :<nl><c-block2><nl>\ti+=1
<if>          ::= if <cond-expr>: <nl><c-block2>
<if2>         ::= if <cond-expr>: <nl><c-block3>
<cond-expr>   ::= <expr> <comp-op> <expr>
<comp-op>     ::= #lesser# | #greater# | == | !=
<assignement> ::= <var-write> = <expr>
<c-block>     ::= <statements>
<c-block2>    ::= <statements2>
<c-block3>    ::= <statements3>
<statements>  ::= <statement><nl>
                | <statement><nl><statement><nl>
                | <statement><nl><statement><nl><statement><nl>
<statements2> ::= <statement2><nl>
                | <statement2><nl><statement2><nl>
                | <statement2><nl><statement2><nl><statement2><nl>
```

References

[1] White, D. R. *et al*. (2012). Better GP benchmarks: community survey results and proposals. *Genet. Program. Evolvable Mach.,14(1),* 3–29.

[2] O'Neill, M., Nicolau, M., & Agapitos, A. (2014). Experiments in program synthesis with grammatical evolution: A focus on integer sorting. *Proceedings of the 2014 IEEE Congress on Evolutionary Computation (CEC)*, 1504–1511.

[3] Gustafson, S., Ekárt, A., Burke, E., Kendall, G., & Ekart, A. (2004). Problem difficulty and code growth in genetic programming. *Genet. Program.*, *5(3)*, 271–290.

[4] Galván-López, E., McDermott, J., O'Neill, M., & Brabazon, A. (2011). Defining locality as a problem difficulty measure in genetic programming.

[5] Kim, K., Kim, M. H., & McKay, B. (2011). Structural difficulty in estimation of distribution genetic programming. *Proceedings of the 13th Annu. Conf. Genet*.

[6] Kinnear, K. E. J. (1994). Fitness landscapes and difficulty in genetic programming. *Proceedings of the First IEEE Conf. Evol. Comput*.

[7] Vanneschi, L., Tomassini, M., Collard, P., & Clergue, M. (2005). A survey of problem difficulty in genetic programming. *Proceedings of the 9th Congress of the Italian Association for Artificial Intelligence, Proceedings*.

[8] Kinnear, K. E. (1993). Generality and difficulty in genetic programming: Evolving a sort. *Proceedings of the 5th International Conference on Genetic Algorithms*.

[9] Hugosson, J., Hemberg, E., Brabazon, A., & O'Neill, M. (2010). Genotype representations in grammatical evolution. *Appl. Soft Comput. J.*,10, 36–43.

[10] Nicolau, M., O'Neill, M., & Brabazon, A. (2012). Termination in grammatical evolution: Grammar design.

[11] Castelli, M., Vanneschi, L., Silva, S., Agapitos, A., & O'Neill, M. (2014). Semantic search-based genetic programming and the effect of intron deletion. I*EEE Trans. Cybern.*, *44(1)*, 103–113.

[12] O'Neill, M., Ryan, C., & Nicolau, M. (2001). Grammar defined introns: An investigation into grammars, introns, and bias in grammatical evolution. *Proceedings of the Genet. Evol. Comput. Conf. -GECCO 2001*.

Hakan Ayral is a guest lecturer in Galatasaray University, Mathematics Department. His research interests include computational mathematics, genetic programming, parallel programming on GPUs and algorithmic information theory.

Songül Albayrak is associate professor in Yıldız Technical University, Computer Engineering Department. Her research interests include computer architecture, parallel architectures and parallel programming, data mining and computer vision. She is the erasmus coordinator of the department.