Noname manuscript No.
(will be inserted by the editor)

Recursion in Tree-based Genetic Programming

Alexandros Agapitos · Michael O'Neill · Ahmed Kattan · Simon M. Lucas

the date of receipt and acceptance should be inserted later

Abstract Recursion is a powerful concept that enables a solution to a problem to be expressed as a relatively simple decomposition of the original problem into sub-problems of the same type. We survey previous research about the evolution of recursive programs in tree-based Genetic Programming. We then present an analysis of the fitness landscape of recursive programs, and report results on evolving solutions to a range of problems. We conclude with guidelines concerning the choice of fitness function and variation operators, as well as the handling of the halting problem.

The main findings are as follows. The distribution of fitness changes initially as we look at programs of increasing size but once some threshold has been exceeded, it shows very little variation with size. Furthermore, the proportion of halting programs decreases as size increases. Recursive programs exhibit the property of weak causality; small changes in program structure may cause big changes in semantics. Nevertheless, the evolution of recursive programs is not a needle-in-a-haystack problem; the neighbourhoods of optimal programs are populated by halting individuals of intermediate fitness. Finally, mutation-based variation operators performed the best in finding recursive solutions. Evolution was also shown to outperform random search.

Alexandros Agapitos
Natural Computing Research and Applications Group, School of Computer Science
University College Dublin, Republic of Ireland
E-mail: alexagapitos@gmail.com

Michael O'Neill
Natural Computing Research and Applications Group, School of Business
University College Dublin, Republic of Ireland
E-mail: m.oneill@ucd.ie

Ahmed Kattan
Computer Science Department, Umm Al-Qura University, Saudi Arabia
E-mail: kattan.ahmed@gmail.com

Simon M. Lucas
School of Computer Science and Electronic Engineering, University of Essex, UK
E-mail: sml@essex.ac.uk

Keywords Evolutionary Program Synthesis, Genetic Programming, Recursive programs, Variation operators, Fitness landscape analysis

1 Introduction

The vast majority of evolved programs by means of Genetic Programming (GP) [24, 32] are based on a functional, expression-tree representation. GP has produced some impressive results but it still has significant problems with scalability. Most programs are of constant time-complexity, rather than being general programs with non-trivial control structures such as conditionals, iteration and recursion. Investigating ways to best evolve recursive programs will arguably expand the class of problems that GP can solve.

Recursion is a powerful *divide-and-conquer* computer programming concept. It solves a problem by reducing it into smaller/simpler instances of itself, solving those instances, and combining the solutions obtained. Most programming languages support recursion by allowing a call to a procedural abstraction (i.e. function or method) to appear in its own body, enabling it to call itself. Every recursive function definition has at least one *base case* and one *recursive case*. A base case deals with input representing the simplest instance of the problem; it provides a solution for that instance and causes the recursive function to terminate. A recursive case deals with any other input, and its purpose is to reduce intermediate problem instances towards the base case. The representation power of recursion allows for the synthesis of a particular type of recursive program, called *tail-recursive*, which can be directly translated into a program with *iteration*. In tail-recursion the last statement in a function is a recursive call.

1.1 Evolution of recursive programs with tree-based GP

There exist two dominant approaches to the evolution of recursive programs with tree-based GP. The first uses explicit recursive calls that enable an evolved program to call itself. All that is required is a recursive primitive in the function-set, whose signature (i.e. return type and parameter types) is the same as that of the evolved program. After evaluating the arguments of a recursive tree-node the evaluation flow is transferred to the root of the expression-tree. Importantly, an explicit base case is required to be evolved as part of the evolved program in order for it to halt.

The second approach is based on higher-order functions that provide an implicit, bounded form of recursion [1, 5, 27, 45–47]. A higher-order function requires two arguments: a data-structure and an operation. It recursively processes the constituent elements of the data-structure through the use of the combining operation, building up a return value. The code-content of the combining operation is subject to evolution. The advantage of this type of building-block is that it encapsulates the recursive execution, and therefore the overall evolved program does not require an explicit base case in order to terminate; recursion is automatically terminated when all elements of the data-structure have been processed, and execution flow is transferred back to the node within the expression-tree where the higher-order function is rooted. Higher-order functions are a powerful addition to standard GP, which enables the evolution of programs with greater than constant-time complexity, although the evolved programs are not actually recursive programs. Higher order functions are similar to *bounded iteration schemata* of [17] and `for-each-loop` constructs of [10, 11]. Their main disadvantage is that they require a significant amount of domain knowledge since different problems have different repetition patterns, and their solution requires different higher-order functions.

Higher-order functions are only applicable to a limited number of problems. Not every solution, which is effectively represented with explicit recursive calls, can be represented using higher-order functions. Evolving recursive programs with explicit recursive calls is arguably the most general and expressive approach between the two. Certainly, higher-order functions can be used to complement the expressiveness of programs with explicit recursion, and this was shown to be fruitful in the evolution of efficient sorting algorithms [1]. However, it is the evolutionary synthesis of computer programs with explicit recursive calls that is our focus here.

1.2 The troubling aspects of the space of recursive programs

The space of recursive programs is fundamentally different from the space of Boolean or symbolic regression programs with constant time-complexity. The major difference is the presence of *non-halting* programs. The two requirements for a recursive program to halt is a base case, as well as the convergence of program arguments (during successive recursive calls) towards that base case. The halting problem presents a stumbling block for the effective evolution of recursive programs. Research has shown that in Turing-complete fitness landscapes (linear programs with finite memory, conditional branches and jumps) the proportion of halting programs falls towards zero with increasing program size [23, 25]. This confirms the belief that Turing-complete program spaces are hard to search. The halting problem also raises the question of how to assign fitness to a program which may recurse indefinitely.

In addition, it has been argued [8, 42, 45] that the difficulty of evolving recursion is associated with the *weak causality* of recursive programs. The property of *causality* relates changes in the structure of an object with the effect of these changes in the behaviour of the object [33]. Specifically, evolved recursive programs are believed to be very sensitive to the application of variation operators, meaning that small changes in the structure of an expression-tree may have a large effect on program semantics and therefore fitness, which may mislead the search process. Fitness functions may be deceptive in the sense that they do not reflect the structural proximity of an individual to a solution. In a preliminary study [5], random walks were used to study the landscape of programs composed of higher-order functions, and therefore guaranteed to halt. Results demonstrated that the landscape is difficult to search since a single random step is very often adequate to severely degrade program fitness.

Several authors [28, 29, 40] have discussed the difficulty of designing a fitness function able to partially reward a recursive program. An optimal program requires both base and recursive cases, as well as their appropriate interplay, and it is unclear how to penalise non-optimal individuals that contain either of these components but not both. For example, a program that contains a correct recursive case but misses the base case will result in infinite recursion, and will often be heavily penalised as it returns an error for any input case. This may disorient evolution, and may result in premature elimination of programs with standalone recursive cases from the population in case where low mutation probability is used [2].

Finally, a discussion of the differences between the topologies of the space of functions (programs of constant time-complexity) and the space of algorithms (programs of higher than constant time-complexity and state variables) is presented in [37]. Standard GP works by searching syntactically nearby algorithms, and relies heavily on the notion that if an algorithm is successful then other algorithms "near" it in the space of algorithms will have a higher chance of being successful than algorithms chosen at random from the space. However, when a program representing an algorithm is changed in some small way, the effect of that change in semantics is compounded on each new iteration of the program's execution.

1.3 Explicit recursion in Genetic Programming: A Survey

Koza (1992) [19] (chapter 18.3) was the first to study a very limited form of recursion for evolving a program that generates the Fibonacci sequence. The evolved program was given ordered input values of the sequence, and made use of a *sequence referencing function* to access previously computed values in the sequence. Explicit recursive calls were not allowed in this work.

Nordin and Banzhaf (1995) [30] evolved Turing-complete programs for a register machine using linear GP. The system enabled recursion synthesis by having local registers initialised to an evolved Automatically Defined Function (ADF), enabling the function to call itself.

Brave (1996) [8] evolved recursive programs for tree search. A non-terminal element was made available in the function-set, allowing the evolved function to call itself. Different program architecture setups including basic GP and GP with ADFs were investigated. To handle infinite recursion, the maximum number of allowed recursive calls was limited to the depth of the tree being searched.

Whigham and McKay (1996) [40] tackled the problem of evolving a *member* function. This is a function that takes as input a list and an element and returns `true` if the list contains the element, and `false` otherwise. The work used directed mutation operators, which are both problem specific and incorporate knowledge about the solution. Recursion is enabled via a recursive function-set element. Whigham and McKay were unable to evolve a recursive solution to the problem. They mainly attributed this result to the inefficiency of the fitness function to partially reward correctness, i.e, a program that contains a correct recursive case but misses the base case. Infinite recursion is handled by setting a maximum number of evaluation steps. If the limit is reached, fitness evaluation is abandoned, and an individual is assigned the worst possible fitness value.

Wong and Leung (1996) [43] evolved recursive programs for the Even-*n*-parity problem. A logic grammar in used that includes a rule that enables recursive calls, and another rule that enforces a termination condition. However, the convergence of the arguments of recursive calls towards the termination condition is not guaranteed, and a recursion limit is put in place to terminate non-halting programs. Wong and Leung demonstrated that their system was able to find the solution to the general Even-*n*-parity problem more efficiently than Koza's ADF approach [20]. In another study, Wong and Leung [44] tackled the evolution of Even-*n*-parity programs from noisy examples. Three experiments were performed to study the impact of noise in training examples on the speed of learning. The study confirmed that GP can evolve recursive solutions from noisy training observations, and that the amount of computation correlates positively with the level of noise.

Huelsbergen (1997) [15] evolved machine-language programs for generating the sequences of square, cube, Factorial, and Fibonacci numbers. Evolved programs maintain a *program counter*. This is an integer that selects an instruction to fetch and execute. Four branching (jump) instructions modify the program counter to point to a new instruction, and have the ability to emulate recursive calls by transferring the execution flow to the start of an evolved program. Huelsbergen demonstrated the effectiveness of a hybrid local/global search system (crossover and hill climbing) over mere random search. The halting problem is handled by placing a limit to the number of instruction executions. The interpreter evaluates a n-instruction program with respect to an m-register input state for a number of evaluation steps. An `Out` instruction appends the value of its register argument to output stream that forms the basis for computing fitness.

Nishiguchi and Fujimoto (1998) [29] proposed two GP systems with *niching* for evolving programs that generate the Fibonacci sequence, and programs for searching maze-like maps. The study argues that diversity is hard to maintain during recursive program evolution mainly due to the inability of the fitness function to assess partially-correct programs. This impairs the continuity of the fitness landscape and makes evolution susceptible to local optima. In the proposed method, three different niches in the form of sub-populations are maintained: one with non-recursive individuals (niche 1), one with non-halting recursive individuals (niche 2), and one with halting recursive individuals (niche 3). Selection is performed on a per-niche basis, while crossover is performed between individuals of different niches. Recursion is implemented using a recursive function-set element. Non-halting individuals in niche 2 are assigned the average fitness of individuals in niches 1 and 3.

Clark and Yu (1999) [13] evolved recursive programs for the *mapcar* and *nth-3* functions. The mapcar function takes two arguments, a function F and a list L, and returns a list obtained by applying F to each element of L. The *nth-3* function takes two arguments, an integer N and a list L, and returns the N^{th} element of L. If $N < 1$, it returns the first element of L. If $N > length(L)$, it returns the last element of L. Recursion is enabled via a recursive function-set element. The fitness function explicitly penalises two types of run-time errors. The first is caused by non-terminating recursion, and the second is caused by taking the `car` or `cdr` of an empty list. A scalar value, which is either constant or a function of the length of the argument list, is subtracted from the overall fitness of an individual in case of run-time errors. The number of recursive calls is limited by the length of the argument list. If the limit is reached, evaluation is abandoned and the run-rime error is flagged.

Koza et al. (1999) [21] (Chapter 8, page 147) introduced Automatically Defined Recursion (ADR). There are four components to ADR, namely, recursion condition (RCB), recursion body (RBB), recursion update (RUB), and recursion ground branch (RGB). To prevent infinite recursion a limit is placed on the number of the recursive calls that are allowed within an individual. When the limit is reached the fitness of the individual is heavily penalised. The RBB branch normally contains a recursive call to ADR, and when the evaluation of RBB finishes, the evaluation is transferred to the RUB branch. The RCB branch determines if recursion is continued by returning a positive value. In the case of returning a negative value recursion is halted and the RGB branch gets evaluated.

Kochenderfer (2003) [18] worked on a block-stacking problem, and demonstrated how GP may be used to evolve hierarchical/modular recursive programs. A recursive function-set element is used to enable recursion. An individual is allowed a maximum number of recursive calls that varied with the number of blocks in the problem. If the limit is reached, evaluation is abandoned.

Spector, Klein, and Keijzer (2005) [36] introduced `Push3` as a language for evolving programs with complex control structures. The problems of list reversal, Factorial, Fibonacci sequence, Even-*n*-parity, and list sorting were tackled in this study. The most significant change in the `Push3` version of the `Push` language is the introduction of the `EXEC` stack, which stores expressions, instructions, and literals for the Push interpreter to subsequently execute. Recursion is implemented using the combinator instruction `EXEC.Y`. Complex computational processes can be built-up from simple expressions on the `EXEC` stack.

`EXEC.Y` inspects the top of the `EXEC` stack, *A*, and then inserts the list `(EXEC.Y A)` as the second item on the `EXEC` stack. This results in an endless recursive call to the unnamed non-recursive function *A*. Recursion may be terminated through further manipulation of the `EXEC` stack using a control manipulation instruction (i.e. a conditional) within *A*. The possibility of unbounded recursive calls is counteracted by imposing execution-step limits (set between 150 and 1,000 for the experiments in [36]). When a program exceeds the execution-step limit a fitness penalty may be imposed. Depending on the problem tackled, a different level of penalty severity is adopted.

Wong (2005) [42] used an adaptive grammar-based genetic programming system (GBGP) to evolve recursive solutions for the problems of Even-*n*-parity and *greater-all*. The *greater-all* function accepts two arguments, a number and a list, and returns `true` if the number is greater than any other number in the list, and `false` otherwise. An *extended logic grammar* differs from a *context-free grammar* in that the grammar symbols, whether terminal or non-terminal, may include arguments. Arguments can be used to enforce context-dependency. Another difference is that an extended logic grammar allows a non-terminal at the right hand-side of a grammar rule to be followed by an optional list of *rule-biases*, which is a list of pairs. The first element of a pair is a number that identifies a grammar rule while the second element of a pair is an integer between *min-rule-bias* and *max-rule-bias*. This integer specifies the relative probability of applying the corresponding grammar rule to expand the non-terminal symbol.

Recursion in GBGP is enabled via the use of a non-terminal symbol that allows the evolved program to call itself. A recursion limit is enforced to avoid the problem of infinite recursion. A program is terminated if after invoking it recursively 20 times it fails to return a result. In that case, it gets assigned a special fitness value. Two techniques are implemented in adaptive GBGP, which allow extended logic grammars to be modified dynamically when offspring are created by crossover operation. The first one attempts to increase/decrease the probability of generating good/poor programs. The second one tries to reduce the chance of creating non-terminating programs. The grammar is automatically modified after observing a number of non-terminating programs. It was found that adaptive GBGP reaches a higher probability of success than a standard GBGP introduced earlier in [43]. Additionally, in the case of adaptive GBGP, the modified grammars reduced the rule-biases of certain rules in order to decrease the chance of generating non-terminating programs.

Agapitos and Lucas (2006) [2] proposed an Object-Oriented GP (OOGP) system to evolve recursive programs for Factorial, Fibonacci, Exponentiation, Even-*n*-parity and *Nth-3* (for problem description see [13]). Recursion is enabled via a recursive function-set element, and a limit of 288 recursive calls is imposed during program evaluation. When evaluation reaches the recursion limit, the program is assigned the worst possible fitness value. The paper studied the efficiency of subtree crossover and subtree mutation operators in discovering solutions, and concluded that a higher mutation probability is more successful in learning recursive programs. This was mainly attributed to the fact that the majority of programs containing a recursive node, which were sampled during early generations, are non-terminating ones and consequently have very low fitness. This causes a premature elimination of recursive structures in cases where search heavily relies on crossover operator. In addition, *random search* was shown to be outperformed by evolution, however random search yielded a probability of success of 6% for Factorial, 2% for Exponentiation, 1% for Even-*n*-parity, and 2% for *Nth-3*. Fibonacci could not be solved using random search.

Agapitos and Lucas (2006) [1] further applied OOGP to evolve recursive sorting algorithms of $O(n \times log(n))$ time-complexity. Higher-order functions, offering implicit recursion, were combined with explicit recursive calls. A upper bound was set to the permissible recursive calls (i.e. 10,500), after which evaluation is abandoned and individuals are assigned the worst possible fitness. Five measures of sequence disorder were studied as fitness functions, with the one defined as the average distance that elements must travel in order to reach their sorted position performing the best.

Agapitos and Lucas (2007) [4] continued work in evolving hierarchical/modular recursive programs, and managed to evolve modular recursive sorting algorithms of $O(n^2)$ time-complexity. The time-complexity of sorting was worse than that of the previous study by Agapitos and Lucas in [1], however the main aim of the work in [4] was to demonstrate that modular recursive programs can be successfully evolved. The modular architecture of the sorting algorithms was based on ADFs, which were themselves allowed to be recursive. Five fitness functions introduced in [1] were compared, with the *mean sorted position distance* performing the best. Furthermore, performance differences between subtree crossover and subtree mutation showed that mutation outperformed crossover. In addition, mutation was shown to produce more parsimonious solutions that crossover. Random search was unable to discover any solutions.

Agapitos and Lucas (2007) [3] evolved modular recursive programs for calculating the mean, variance, and standard deviation of a list of real numbers, as well as the length of the list of numbers. Each operator was represented using a separate ADF as part of a multi-ADF program architecture. ADFs were allowed calls to themselves and to each other, and they were cooperatively coevolved in the same run. The maximum allowed number of recursive calls in each ADF was limited to the length of the input list. Non-halting programs were assigned the worst possible fitness.

Wilson and Heywood (2007) [41] introduced Probabilistic Adaptive Mapping Developmental Genetic Programming that co-operatively co-evolved a population of adaptive mappings and associated genotypes, which are used to learn recursive solutions of Factorial and Fibonacci. The function set consisted of general machine-language instructions similar to the earlier work of [15]. Recursion is enabled via conditional/unconditional branching instructions that cause the *program counter* to point to their target instruction. A program terminates after running all its instructions or after the execution of 100 steps.

Shirakawa and Nagao (2010) introduced Graph Structured Program Evolution (GRAPE) in [35]. Program representation in GRAPE consists of a directed graph of nodes and indexed memory. Data flows through the directed graph and is processed at each node. There are two types of node; one for processing and one for branching. Cyclic connections between nodes can realise loops, whereas connections from any node to the start node can realise recursive calls. The effectiveness of GRAPE to automatic programming tasks was shown through a series of experiment to evolve programs for Factorial, Exponentiation and Sorting using branching and loops.

Moraglio et al. (2012) [28] experimented with a method, which allows for the separate evolution of recursive and base-cases in a recursive program. Evolution is set to synthesise optimal non-recursive programs out of a special function primitive called a *scaffolding* function. A *scaffolding* function is a tree-node that returns the correct answer for all entries in the training set for the problem. Once a scaffolding-based program passes all training cases, all the occurrences of the *scaffolding* tree-node are replaced with a recursive tree-node (i.e. a function that transfers control to the root of a tree), therefore converting the program into a truly recursive one. As a side benefit, there is no need to take any measures against non-halting individuals during evolution. Two problems are tackled; that of list reversal, and that of inserting an element into an ordered list while maintaining the correct ordering. A search regime based on a blend of crossover and mutation outperformed search regimes solely based on crossover or mutation. Crossover alone was extremely inefficient in discovering solutions in either of the recursive problems. This finding is inline with findings of Agapitos and Lucas [2], which suggested the use of a higher mutation rate in relation to the crossover rate for evolving recursive programs.

Alexander and Zacher (2014) [6] used *call-trees* to guide the evolution of recursive programs in an extended Grammatical Evolution system. A call-tree is used to describe the function-call hierarchy in a program. Each node in the tree contains the parameters of the function call. Each child-node represents the sub-calls made by its parent-node. The proposed method separates evolution of recursive and base cases. The search process consists of two phases. In the first phase, the user-provided call-tree is traversed to create a set of tuples, where the first element of each tuple refers to the input parameter of a parent node, and the second element refers to a list of child-node input parameters. Using this information, the system constructs a grammar that accommodates the number of recursive calls depicted in the call-tree, and then proceeds with the evolution of an expression that evaluates to the recursive parameters required in each recursive call within the program. As an example, for the problem of Factorial, this expression should be `x-1`, where x is the input variable. The set of tuples generated from the call-tree are used as the training data for this step. In the second phase, the previously evolved code is embedded in the grammar, and a new evolutionary run is launched to synthesise the complete recursive function using a training set of input-output pairs. A number of recursive problems were tackled, and it was found that the proposed method significantly outperformed Grammatical Evolution both in terms of the number of fitness evaluations and the number of times a 100% correct solution was evolved.

Turner and Miller (2014) introduced Recurrent Cartesian GP (RCGP) in [39]. The distinct characteristic of RCGP is the creation of cyclic graphs that enable it to store internal state information. Its implementation allows connections between a given node and any other node in the program including the program itself, thus allows recursive calls. Despite of the fact that the goal of the work in [39] was mainly to assess RCGP performance improvements over standard Cartesian GP in tackling the partially observable tasks of Artificial Ant and Sunspot series forecasting, here we make a note of its potential capability of evolving general recursive programs.

2 Scope of research

Research has shown that recursive programs with explicit recursive calls can be reliably evolved from small samples of training examples, and that these programs generalise perfectly. With the notable exception of the works of [6, 28], the complete structure of the recursive program (i.e. both recursive and base cases) is searched for in an evolutionary run. In order to handle infinite recursion, the vast majority of previous work placed a hard threshold on the number of allowed recursive calls. This was either set equal to the length of the input argument or to some arbitrary number decided by the experimenter. Abandoning the evaluation of a recursive individual is followed by either assigning it the worst possible fitness, or in some cases, allowing for a more fine-grained evaluation based on the progress of the evolved solution up to that point.

Several papers surveyed in the previous section reported on the probability of success and computational effort required to construct recursive solutions. However, most of the arguments raised about the properties of the space of recursive programs are currently qualitative. Here we bridge this gap and present an in-depth analysis of tree-based GP for the evolution of recursive programs. We first use a series of techniques to analyse the fitness landscape, and then present simulation results from evolving solutions to a range of recursive problems. The research questions that we address are summarised bellow:

1. How does the functionality of recursive programs changes with respect to their size? What is the distribution of non-halting programs?
2. Do recursive programs exhibit a weak or strong causality? How is the fitness landscape structured in the neighbourhood of optimal individuals?
3. What is the effect of different variation operator schemes in searching the space of recursive programs? What is the role of diversity in the process, and how are the levels of diversity affected by different variation operators?

The rest of the article is structured as follows. Section 3 starts by introducing the recursive problems. It then presents the fitness landscape analysis techniques, the GP system setup, and the variation operators employed. Section 4 analyses the results, while Section 5 presents guidelines for using recursion in GP, and discusses open issues for further investigation. Section 6 concludes the article.

3 Methods and Experiment design

3.1 Problems

Five different recursive problems are tackled. They are defined as follows:

Factorial

$$Factorial(i) = \begin{cases} 1 & \texttt{if i < 1} \\ i * Factorial(i-1) & \texttt{otherwise} \end{cases} \tag{1}$$

Fibonacci

$$Fibonacci(i) = \begin{cases} 1 & \texttt{if i = 0 or i = 1} \\ Fibonacci(i-1) + Fibonacci(i-2) & \texttt{otherwise} \end{cases} \tag{2}$$

Exponentiation

$$Exponentiation(base, exponent) = \begin{cases} 1 & \texttt{if i < 1} \\ base * Exponentiation(base, exponent-1) & \texttt{otherwise} \end{cases} \tag{3}$$

Even-n-parity This problem takes as input a list *l* of N Boolean values, returning `true` if an even number of inputs are `true`, otherwise it returns `false`.

$$Parity(l) = \begin{cases} true & \texttt{if l is empty} \\ (car(l) || parity(cdr(l))) \&\& (!(car(l) \&\& parity(cdr(l)))) & \texttt{otherwise} \end{cases} \tag{4}$$

where *car* returns the first element of a list and *cdr* returns the tail of a list.

Nth This problem takes as input two arguments, an integer n and a list l of doubles, and returns the n^{th} element of the list. If n is less than one, it returns the first element of the list. If n is greater than the length of the list, it returns the last element of the list.

$$Nth(n, l) = \begin{cases} car(l) & \texttt{if n <= 1} \\ Nth(n-1, l) & \texttt{if n > size(l)} \\ Nth(n-1, cdr(l)) & \texttt{otherwise} \end{cases} \tag{5}$$

where *car* returns the first element of a list and *cdr* returns the tail of a list.

Table 1 Strongly-typed language for evolving recursive programs.

FUNCTION SET		
	Argument(s) type	**Return type**
Arithmetic		
add	double, double	double
sub	double, double	double
mul	double, double	double
div (protected)	double, double	double
Boolean logic		
and, or, nand, nor	Boolean, Boolean	Boolean
List processing		
car	list<double>	double
car	list<Boolean>	Boolean
cdr	list<double>	list<double>
cdr	list<Boolean>	list<Boolean>
isEmpty	list<double>	Boolean
isEmpty	list<Boolean>	Boolean
length	list<double>	double
length	list<Boolean>	double
Recursive nodes		
Factorial	double	double
Fibonacci	double	double
Exponentiation	double, double	double
Even-n-parity	list<Boolean>	Boolean
Nth	list<double>, double	double
Predicates		
$>, \geq, =, <, \leq$	double, double	Boolean
Conditional		
IF-Then-Else	Boolean, double, double	double
IF-Then-Else	Boolean, Boolean, Boolean	Boolean
TERMINAL SET		
	Value	**Type**
Constants	$\{-10.0, -9.0, -8.0, \ldots, 8.0, 9.0, 10.0\}$	double
Parameters	training/test input value	double list<double> list<Boolean>

3.2 Language for constructing recursive programs

We defined a general-purpose, strongly-typed primitive language for constructing and evolving recursive programs, presented in Table 1. Recursion is implemented using explicit recursive calls realised by tree-nodes that transfer the flow of control to the root of the expression-tree. Base cases for terminating recursion can be synthesised using the `If-Then-Else` primitive.

The function sets for different problems are illustrated in Table 2. The terminal sets consist of constants in Table 1, as well as input parameters according to the problem. In the problem of Even-*n*-parity the primitive `car` returns a value of type `Boolean`. In the same problem, the primitive `IF-Then-Else` accepts 3 arguments of type `Boolean`, `Boolean`, `Boolean`, and returns a value of type `Boolean`. In the problem of Nth, `car` returns a value of type `double`. In this case, the primitive `IF-Then-Else` accepts 3 arguments of type `Boolean`, `double`, `double`, and returns a value of type `double`.

3.3 Distribution of program functionality

This part of fitness landscape analysis investigates how the functionality of programs changes with respect to their size for a given problem. In addition, we study how the proportion

Table 2 Function sets for different problems.

Problem	Input type(s)	Return type	Function set elements
Factorial	double	double	arithmetic, predicates, conditional, recursive node
Fibonacci	double	double	arithmetic, predicates, conditional, recursive node
Exponentiation	double, double	double	arithmetic, predicates, conditional, recursive node
Even-*n*-parity	list<Boolean>	Boolean	Boolean logic, predicates, list processing, conditional, recursive node
Nth	list<double>, double	double	arithmetic, list processing, predicates, conditional, recursive node

of halting programs is distributed with increasing program size. We used the *ramped uniform initialisation* method presented in [22] to sample the program space (C++ code can be found at `ftp://ftp.cs.ucl.ac.uk/genetic/gp-code/rand_tree.cc`). We then plotted the proportion of programs of a given size (measured in terms of number of tree-nodes) by their error (i.e. value of loss function), in order to study the distribution of functionality and the distribution of non-halting programs for a given recursive problem.

For each program size in the interval of $\{5, \ldots, 60\}$, we sampled and evaluated 2,000,000 programs. Each program is assigned to an *error-class*, and proportions are calculated as the ratio of the number of programs belonging to a particular class over the total number of programs sampled. In the case of Even-n-parity which uses a discrete loss function, the determination of fitness classes is straightforward; we assign each error-class to every possible value of the loss function. For the rest of the problems that use continuous loss functions, discretisation needs to be performed in order to assign a program to an error-class. In our experiments, continuous loss functions are normalised within the $\{0.0, \ldots, 1.0\}$ interval. This interval is further divided into 100 discrete error-classes using a step of size 0.01. Program error is rounded to the closest discrete error-class.

3.4 Random walks and Error-Distance correlation analysis

Random walks are employed to study the type of *causality* in recursive program representations, and to investigate how the fitness landscape is structured around optimal individuals. For each evolved solution considered, $1,000$ random walks of length 20 are performed using the standard subtree mutation operator (Section 3.5.4). In every problem we used a range of successfully-evolved programs with various size and depth levels.

Random walks are instrumented in order to:

1. Plot the error (i.e. value of loss function) versus the *edit-distance* from the optimum program.
2. Calculate the Pearson correlation coefficient between error and edit-distance.
3. Measure the proportion of single-step walks (i.e. a neighbour reached by a single mutation on a perfect program) that result in a non-halting program.
4. Measure the average walk-length after which a perfect individual becomes a non-halting program and their edit-distance at that point.

The version of the edit-distance that we used is defined in [14] as follows. Two trees are brought to the same structure by adding `null` nodes to each tree. A constant is set to weight the depth of tree differences differently. The distance between two nodes p and q is given by:

$$d(p,q) = \begin{cases} 1 \texttt{ if they are equal nodes} \\ 0 \texttt{ otherwise} \end{cases} \quad (6)$$

The distance between two trees T_1 and T_2 is given by:

$$dist(T_1,T_2) = \begin{cases} d(p,q) & \texttt{if } T_1 \texttt{ and } T_2 \texttt{ are leaves} \\ d(p,q) + K * \sum_{i=1}^{m} dist(s_i,t_i) & \texttt{otherwise} \end{cases} \quad (7)$$

where T_1, T_2 are trees with roots p and q and possible m subtrees s, t. The constant K is set to 0.5 as in [14].

3.5 Evolving recursive programs

The final part of our analysis compares a range of variation operators in terms of their efficiency at discovering solutions. For each problem and each variation operator, we performed 100 independent evolutionary runs, and reported the probability of success. We also studied the evolution of phenotypic diversity maintained by different variation operators. Diversity is measured in terms of fitness *entropy* [9,34].

3.5.1 Training and Test Data

The problem of learning recursive programs is formulated as the general function estimation problem: we are given a set of training examples $\{(x_i, y_i)\}_{i=1}^{N}$, where $y \in \mathbb{R}$ is the response variable and $\mathbf{x} \in \mathbb{R}^d$ is a vector of explanatory variables. The goal is to find a function $F^*(\mathbf{x})$ that maps $\mathbf{x}$ to y, such that over the joint distribution $P(\mathbf{x}, y)$ the expected value of a loss function $L(y, F(\mathbf{x}))$ is minimised:

$$F^*(\mathbf{x}) = \underset{F(\mathbf{x})}{\arg\min} \mathbb{E}_{x,y}[L(y, F(\mathbf{x}))] \quad (8)$$

In the case of recursive computer programs, the expected value of the loss function is 0.0 for an optimal individual, which is a program that successfully reproduces the exact response variable values in a test set. A set of training examples is used to guide the evolutionary search, and evolution is terminated if a maximum number of generations is reached or an individual achieves the loss function value of 0.0 in the training set. Generalisation is measured on an independent set of test cases, and recognises the success of a run. Table 3 presents the training and test datasets.

3.5.2 Loss Functions

For Factorial, Fibonacci and Exponentiation problems, the loss function takes the form of normalised mean absolute error known as *Canberra distance*. The normalisation gives equal weight to the errors resulting from each training case by placing each error in the $\{0.0, \ldots, 1.0\}$ interval. This restricts larger errors from dominating the mean error, which is important for problems such as Factorial. The loss function is defined as follows:

Table 3 Training and Test data for different problems.

Problem	**Training Data**	**Test Data**
Factorial	10 examples, input x $\in \{1, \ldots, 10\}$	10 examples, input x $\in \{11, \ldots, 20\}$
Fibonacci	12 examples, input x $\in \{1, \ldots, 12\}$	10 examples, input x $\in \{13, \ldots, 22\}$
Exponentiation	10 examples of 2^n, $n \in \{0, \ldots, 9\}$	10 random examples of m^n $m, n \in \{0, \ldots, 10\}$
Even-*n*-parity	12 examples of Even-2-parity	128 examples of Even-7-parity
Nth	20 examples list of unique elements $\{x_i\}_{i=1}^{20}$, $x \in [1.0, 100.0]$ 20 values of input $n \in \{-4, \ldots, 15\}$	30 examples list of unique elements $\{x_i\}_{i=1}^{30}$ 30 values of input $n \in \{-4, \ldots, 25\}$

$$L(y, F(x)) = \frac{|y - F(x)|}{|y| + |F(x)|} \tag{9}$$

where $|\cdot|$ returns the absolute value, y is the target response variable, and $F(x)$ is the output of an evolved program F given input vector x. For Factorial and Fibonacci, the input vector is of a single dimension, whereas for Exponentiation the input vector is two-dimensional.

We used a *zero-one* loss function for the Even-*n*-parity problem, where all *misclassifications* are charged one unit. That is:

$$L(y, F(x)) = I(y \neq F(x)) \tag{10}$$

where $I(\cdot)$ is the indicator function, y is the target response variable, and $F(x)$ is the output of an evolved program F given input vector x. The input vector in this case is a list of Boolean variables, and the response variable is a Boolean variable.

For the Nth problem, the loss function is defined as follows:

$$L(y, F(x)) = 1 - \frac{1}{2^{d(y, F(x))}} \tag{11}$$

where $d(y, F(x))$ is the distance (within the list) between the correct position of element y and the position of the program return-value $F(x)$ given input vector x. The loss function's output is constrained within the $\{0.0, \ldots, 1.0\}$ interval. In cases where $F(x)$ does not match an element of the input list then the maximum distance of 1.0 is returned.

The average value of a loss function on N training examples is simply:

$$\frac{1}{N} \sum_{i=1}^{N} L(y_i, F(x_i)) \tag{12}$$

3.5.3 Handling infinite recursion

To avoid the problem caused by non-terminating recursive programs we imposed a limit on the allowed number of intermediate recursive calls. This limit was set to $10,000$ recursive calls. When a program reaches the recursion limit we assume that the program is a non-halting one, evaluation is abandoned and the individual is assigned the worst possible fitness according to the problem. For Factorial, Fibonacci, Exponentiation, and Nth, a non-halting individual is assigned the fitness value of 1.0. For the problem of Even-*n*-parity a non-halting individual is assigned the fitness value of 12.

3.5.4 Variation operators and run parameters

We contrasted the search performance of 4 different variation operator schemes, and also that of random search. These are described bellow.

Subtree recombination + Subtree mutation (SR+SM). This is the traditional search regime introduced in [19], which relies heavily on recombination using standard subtree crossover. While the work of [19] did not use any mutation, we deliberately employ subtree mutation, which is applied with a very low probability as opposed to crossover. The parameterisation of the operators is given in Table 4.

Subtree mutation (SM). This is a search regime that relies solely on mutation, and makes no use of recombination. Under type and depth constraints a node is selected with uniform probability and the subtree rooted at that node is replaced by a newly generated subtree of the same type. The tree-generation procedure is *grow* or *full* [19]. To improve on the exploratory capability of the mutation operator, other than picking the tree-node to be replaced from the whole expression-tree, we devised an additional node-selection method. In this method a depth-level is picked uniform-randomly from the range of all possible depth-levels present in the expression-tree, and subsequently a node is picked uniform-randomly from the set of nodes that lie in the chosen depth-level. The decision between the two node-selection methods is governed by a probability set in favour of the uniform-random selection from the whole expression-tree.

Multi-mutation (MM). This operator is introduced in [12], and is designed to counteract the undesirable tendency of traditional subtree crossover and mutation operators to affect smaller and smaller fractions of the expression-tree as the tree grows larger. Multi-mutation enables the transition from one location to any other location of the search space in a sequence of finite steps. It is a composite operator formed using a set of 6 distinct mutation types: *one-node*, *all-nodes*, *swap*, *grow*, *trunc*, *gaussian*; their description can be found in [12]. This operator proceeds as follows: First a sample n from a Poisson random variable with certain mean λ is drawn. Then, n mutation types are uniform-randomly picked with replacement from the set. Each of these mutation operators is applied in sequence, in a pipe-and-filter manner in order to generate a new individual. As an example, given parent p, $n = 2$, and a sequence of mutation types $(oneNode, grow)$ then offspring $o = grow(oneNode(p))$.

Uniform recombination + Point mutation (UR+PM). Uniform crossover works in direct analogy to the operator defined for GAs, and it was introduced in GP in the work of [31]. Its application starts by jointly traversing the two trees in a top-down fashion in order to define a *common region*, which contains any node in one parent having a corresponding node at the same location in the other parent. Offspring are created by visiting nodes in the common region and flipping a coin at each locus to decide whether the corresponding offspring node should be picked from the first or the second parent. If the node to

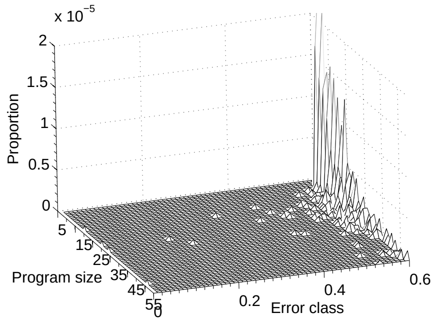
Table 4 Parameterisation of variation operators.

Scheme 1: Subtree recombination + Subtree mutation		
Subtree crossover	prob. of selecting a function	90%
	probability of selecting a terminal	10%
	probability of applying the operator	90%
Subtree mutation	probability of selecting a node from a depth-level	30%
	probability of selecting a node from the whole tree	70%
	depth of subtree (subject to max. depth)	uniform-randomly within $\{1, \ldots, 5\}$
	tree-generation method	50% grow, 50% full
	probability of applying the operator	10%
Scheme 2: Subtree mutation		
	probability of selecting a node from a depth-level	30%
	probability of selecting a node from the whole tree	70%
	depth of subtree (subject to max. depth)	uniform-randomly within $\{1, \ldots, 5\}$
	tree-generation method	50% grow, 50% full
	probability of applying the operator	100%
Scheme 3: Multi-mutation		
	Mutation types	AllNodes, OneNode, Swap, Grow, Trunc
	Poisson λ	1.2
	probability of applying the operator	100%
Grow	probability of selecting a node from a depth-level	30%
	probability of selecting a node from the whole tree	70%
	depth of subtree (subject to max. depth)	uniform-randomly within $\{1, \ldots, 5\}$
	tree-generation method	50% grow, 50% full
Scheme 4: Uniform recombination + Point mutation		
Point mutation	probability of applying the operator	100%
	probability of mutating a node	2 / treeSize

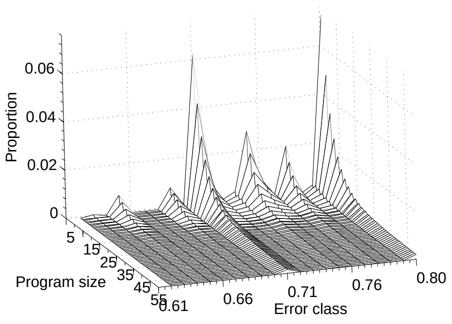
be inherited belongs to the boundary of the common region and it is a function, then the subtree that is rooted there is transferred to the offspring. Point-mutation always follows the application of uniform crossover, with the probability that a tree-node will be mutated set to $2/treesize$. In point-mutation, a tree-node is substituted by another random tree-node of the same type and arity. One of the properties of uniform crossover is that it does not increase the depth of the offspring beyond that of their parents, and therefore beyond the maximum depth of the expression-trees created at the start of evolution.

Random search. The last search regime used is random search (i.e. no selection pressure). This resulted in generations of purely random individuals created with the ramped-half-and-half method with minimum and maximum depths set to 4 and 10 respectively.

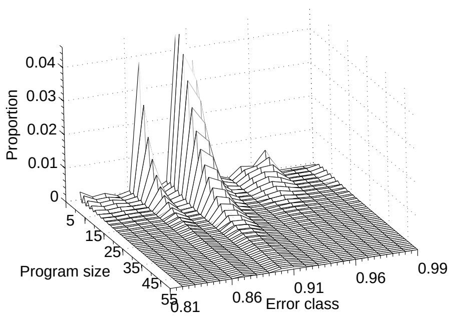
The rest of the evolutionary run parameters are the following. We used an elitist, generational GP system with elitism set to 1 individual. Population size was set to $5,000$ individuals for the problems of Factorial, Even-n-parity, and Nth. The problems of Fibonacci and Exponentiation used a population size of $10,000$. The number of generations was set to 100. Ramped-half-and-half [19] was used to initialise the population, with initial tree-depth ranging from 2 to 5. For the runs of uniform crossover, ramped-half-and-half used tree-depths in the range of 2 to 10. In all problems and variation operator setups, the maximum allowed tree-depth produced during evolution was set to 10. Finally, we used tournament selection with a tournament size of 4.



(a) Factorial. Error interval: $\{0.0, \dots 0.6\}$



(b) Factorial. Error interval: $\{0.61, \dots 0.8\}$



(c) Factorial. Error interval: $\{0.81, \dots 0.99\}$

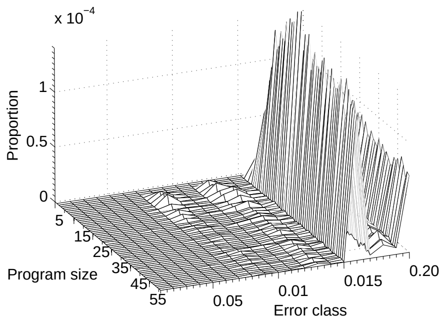
Fig. 1 Proportion of programs (out of 2,000,000 programs) of given error as a function of program size for Factorial. To aid visualisation the distribution of error is presented in consecutive error-class intervals, namely $\{0.0, \dots 0.6\}$, $\{0.61, \dots 0.8\}$, and $\{0.81, \dots 0.99\}$.

4 Results analysis

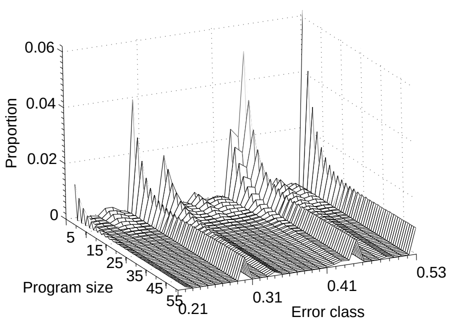
4.1 Distribution of program functionality

Figures 1, 2, and 3 present the distribution of error values as a function of program size, for the problems of Factorial, Fibonacci, and Exponentiation respectively. To aid visualisation, we have divided the distribution and presented it in a number of complementary figures using consecutive error-class intervals. As an example, the distribution of errors for the problem of Factorial is presented in three intervals of $\{0.0, \dots 0.6\}$, $\{0.61, \dots 0.8\}$, and $\{0.81, \dots 0.99\}$ in figures 1(a), 1(b), and 1(c) respectively. In all three problems, the distribution concerns error values in the interval $\{0.0, \dots 0.99\}$. Error of 1.0 is indicative of a non-halting program, and the proportion of non-halting programs as a function of program size is illustrated independently in Figure 4 for all five problems.

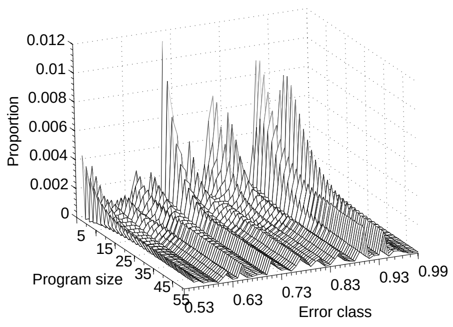
Figures 1, 2, and 3 suggest that, provided that programs exceed some small expression-tree size-threshold, the distribution of error is roughly independent of size. In all three problems, this size-threshold appears to be approximately 30 tree-nodes. In addition, we observe that the proportion of good or near-optimal programs is very small; random programs scarcely cover low error-classes or not at all.



(a) Fibonacci. Error interval: $\{0.0,\ldots 0.2\}$



(b) Fibonacci. Error interval: $\{0.21,\ldots 0.52\}$

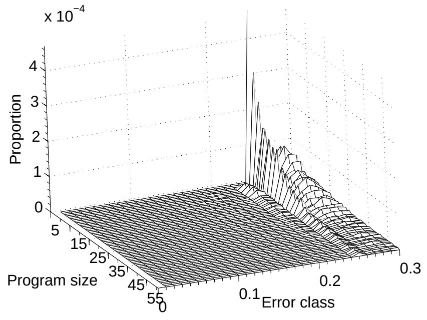


(c) Fibonacci. Error interval: $\{0.53,\ldots 0.99\}$

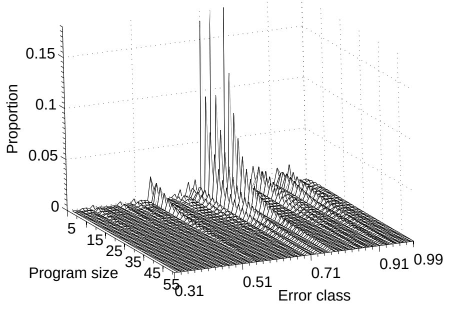
Fig. 2 Proportion of programs (out of 2,000,000 programs) of given error as a function of program size for Fibonacci. To aid visualisation the distribution of error is presented in consecutive error-class intervals, namely $\{0.0,\ldots 0.2\}$, $\{0.21,\ldots 0.52\}$, and $\{0.53,\ldots 0.99\}$.

Figure 4 shows that in all problems the proportion of halting programs falls towards small values with increasing program size. The problems of Factorial and Exponentiation in figures 4(a) and 4(c) illustrate a rapid convergence of this proportion towards the value of zero as a function of size. The problem of Fibonacci in Figure 4(b) shows convergence of proportion of halting programs around 17%. The problems of Even-n-parity and Nth in Figures 4(d) and 4(e) do not show a convergence of the proportion to a particular value, at least for the program sizes studied here.

Overall, our findings are in accordance with simulation results published in [25], which confirmed that in Turing-complete fitness landscapes the proportion of halting programs falls towards zero with increasing program size. The present study concerns small programs of maximum tree-size of 60 nodes. This part of recursive program spaces is populated with a decreasing small fraction of usable programs. We expect that this finding will generalise to larger tree-sizes as well. Specifically, we expect the number of halting programs to rise exponentially with size, but to get increasingly rare as a fraction of all programs as in [25].

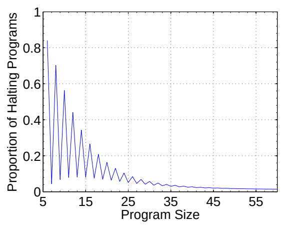


(a) Exponentiation. Error interval: $\{0.0, \ldots 0.3\}$

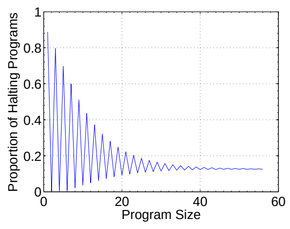


(b) Exponentiation. Error interval: $\{0.31, \ldots 0.99\}$

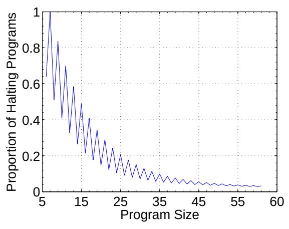
Fig. 3 Proportion of programs (out of 2,000,000 programs) of given error as a function of program size for Exponentiation. To aid visualisation the distribution of error is presented in consecutive error-class intervals, namely $\{0.0, \ldots 0.3\}$, and $\{0.31, \ldots 0.99\}$.



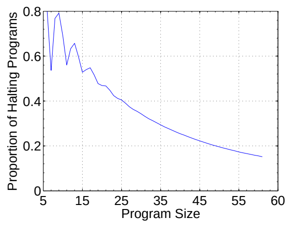
(a) Factorial



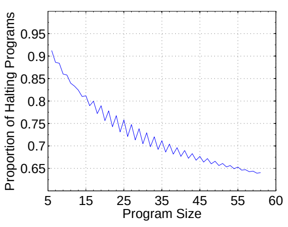
(b) Fibonacci



(c) Exponentiation



(d) Even-n-Parity



(e) Nth

Fig. 4 Proportion of halting programs (out of 2,000,000 programs) as a function of program size.

4.2 Error-distance-correlation and random walk analysis

Figure 5 illustrates scatter plots of error versus edit-distance resulted from random walks with evolved solutions of ascending size highlighted in grey in Table 5. Error refers to the value of the loss function. Those points on the level of maximum error are indicative of non-halting programs, whereas all other points represent halting programs. Table 5 reports the Pearson correlation coefficient between edit-distance and error, as well as the results of random walks using as starting point different successfully evolved individuals of various size and depth levels.

Looking at the Pearson correlation coefficient between edit-distance and error in Table 5, we observe that there is a weak correlation for the problems of Factorial, Fibonacci, Exponentiation, and Even-*n*-parity. The correlation is stronger in the case of Nth. The scatter plots of Figure 5 show that in all problems very small edit-distances from an optimal program are seen with a wide range of error values. In the extreme cases, small edit-distances are coupled with the largest possible error that is indicative of a non-halting program. Specifically, an average of approximately 1/3 of 1,000 single-step neighbours (column v of Table 5) will produce a non-halting program for the problems of Factorial, Fibonacci, Exponentiation, and Even-*n*-parity. The evolved solutions for Nth appear to be more resilient under the application of mutation; an average of less than 1/5 of 1,000 single-step neighbours are non-halting programs.

Recursive program spaces may be difficult to search. Very small variations in the structure of a halting program (as measured by edit-distance) are able to significantly degrade its semantics, and in the worst case result in a non-halting program. On the positive side, non-halting programs are on average three to seven mutation steps away from optimal solutions in all problems considered. The resulting edit-distances at that point similarly range on average within the $\{3.0, \dots, 7.0\}$ interval. The scatter plots in Figure 5 show that the neighbourhoods (programs of small edit-distance from the optimum) of optimal programs are composed of many halting programs of intermediate error. This suggests that the evolution of recursive programs is a difficult but not a needle-in-a-haystack problem, and that there exist a number of mutation-based transitions that enable hill-climbing in this program space.

A final observation in Figure 5 concerns the neutrality of the program space. In all problems there exist a number of neutral mutations where a positive edit-distance is seen with an error of zero. There is a weak trend towards higher neutrality in bigger tree-sizes for the problems of Even-*n*-parity and Nth.

4.3 Evolutionary runs analysis

Figure 7 presents the evolution of best-of-generation error (interpreted as “learning curves”), while Figure 8 presents the evolution of error entropy for all problems using different variation operator setups. Table 6 summarises the probability of success, while Figure 6 illustrates the cumulative probability of success.

A clarification concerning the learning curves needs to be made at this point before proceeding to the discussion of results. The problem of learning general recursive programs is reduced to that of function estimation, in which a response variable needs to be predicted with zero loss. All problem definitions but Even-*n*-Parity have real-valued response variables. The fact that function sets are composed of arithmetic operations allows evolutionary search to sample non-recursive programs in an attempt to minimise the loss function over the training observations. It is interesting to count the number of runs that resulted in recursive programs as opposed to runs that resulted in programs of constant time-complexity, nonetheless this is left as future exercise.

The tendency of the learning curves of certain problems to yield low error levels does not necessarily reflect the probability of finding a solution. For example, in the following sections we shall observe that the evolutionary runs of Fibonacci and Exponentiation attain lower error levels that the evolutionary runs on Factorial. This is mainly attributed to the large differences in the scale of the response variable values that need to be predicted in each problem. In particular, Factorial runs learn the sequence $\{1, 2, 6, \dots, 3, 628, 800\}$,

Table 5 Summary of random walk analysis. Statistics are calculated on 1,000 random walks of length of 20 steps starting from successfully-evolved programs of various shapes (different size and depth levels). Standard error of the mean in parentheses. Single-step random walks in column v refer to neighbours reached by a single application of subtree mutation to a perfectly-evolved program. Columns v, vi, vii report mean values of their constituent elements in bold font. The programs used as optima in the scatter plots of Figure 5 are highlighted in grey.

Evolved solution	**Size** (# nodes)	**Depth**	**EDC Pearson coefficient**	**Proportion of single-step random walks that result in non-halting program**	**Mean random walk length that results in non-halting program**	**Mean edit-distance of non-halting programs from solution**
(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)
Factorial	15	5	0.302	0.34	3.11 (0.07)	2.85 (0.05)
	17	4	0.280	0.34	3.33 (0.08)	2.86 (0.04)
	23	6	0.504	0.23	4.34 (0.09)	3.80 (0.06)
	47	7	0.448	0.22	4.18 (0.08)	4.63 (0.07)
	72	10	0.542	0.11	6.92 (0.12)	6.60 (0.11)
	64	10	0.261	0.37	2.61 (0.05)	2.77 (0.07)
	83	10	0.501	0.14	6.20 (0.11)	5.99 (0.12)
	108	10	0.434	0.20	4.29 (0.08)	5.00 (0.11)
	143	10	0.348	0.22	4.10 (0.08)	4.05 (0.11)
	183	10	0.293	0.32	2.76 (0.05)	3.23 (0.10)
				mean: 0.25	**mean: 4.18**	**mean: 4.28**
Fibonacci	31	6	0.416	0.44	2.33 (0.05)	3.21 (0.09)
	70	7	0.511	0.36	2.78 (0.06)	5.38 (0.14)
	246	8	0.582	0.34	3.04 (0.07)	6.65 (0.17)
				mean: 0.38	**mean: 3.05**	**mean: 5.08**
Exponentiate	21	5	0.424	0.34	3.11 (0.07)	3.20 (0.05)
	31	6	0.432	0.27	3.43 (0.07)	4.37 (0.10)
	67	8	0.439	0.22	4.21 (0.08)	5.07 (0.10)
	73	10	0.342	0.35	2.91 (0.06)	3.40 (0.06)
				mean: 0.29	**mean: 3.41**	**mean: 4.01**
Even-*n*-parity	29	7	0.347	0.26	3.66 (0.08)	3.00 (0.05)
	45	10	0.421	0.24	3.74 (0.07)	2.88 (0.05)
	59	8	0.278	0.3	3.23 (0.07)	2.85 (0.05)
	63	9	0.361	0.33	2.87 (0.06)	3.73 (0.06)
	81	9	0.347	0.26	3.59 (0.07)	3.15 (0.06)
	94	10	0.467	0.30	3.03 (0.06)	2.22 (0.04)
	112	10	0.346	0.20	4.26 (0.08)	2.53 (0.05)
	248	10	0.512	0.20	4.22 (0.08)	4.02 (0.08)
	296	10	0.336	0.33	2.87 (0.06)	3.73 (0.06)
	447	10	0.526	0.25	3.61 (0.07)	3.66 (0.08)
				mean: 0.27	**mean: 3.5**	**mean: 3.20**
Nth	19	4	0.699	0.13	7.72 (0.14)	5.72 (0.06)
	27	5	0.579	0.28	4.82 (0.11)	4.37 (0.09)
	38	10	0.554	0.15	6.28 (0.12)	4.49 (0.12)
	44	8	0.604	0.14	7.23 (0.13)	5.79 (0.11)
	46	7	0.614	0.13	7.37 (0.13)	6.35 (0.11)
	57	7	0.626	0.14	7.04 (0.13)	6.88 (0.12)
	74	7	0.574	0.17	6.30 (0.13)	6.34 (0.12)
	111	7	0.608	0.13	7.22 (0.13)	6.81 (0.11)
	138	10	0.670	0.07	10.09 (0.15)	10.59 (0.16)
	222	10	0.602	0.06	10.48 (0.15)	9.91 (0.15)
				mean: 0.14	**mean: 7.45**	**mean: 6.72**

Table 6 Summary for probability of success (out of 100 independent runs) using different variation operator schemes. Standard errors in parentheses. Best performance highlighted in grey.

	SR+SM	**SM**	**MM**	**UR+PM**	**Random Search**
Factorial	28% (0.04)	52% (0.04)	59% (0.04)	4% (0.01)	10% (0.03)
Fibonacci	1% (0.009)	1% (0.009)	3% (0.01)	0% (-)	0% (-)
Exponentiation	1% (0.009)	4% (0.01)	5% (0.02)	3% (0.01)	2% (0.01)
Even-n-parity	10% (0.03)	9% (0.02)	21% (0.04)	0% (-)	1% (0.009)
Nth	27% (0.04)	15% (0.03)	42% (0.04)	1% (0.009)	2% (0.01)

whereas Fibonacci runs learn the sequence $\{1, 2, 4, \ldots, 512\}$, and Exponentiation runs learn the sequence $\{1, 2, 3, \ldots, 233\}$.

We also found that SR+SM attained lower error levels than those variation operator setups with heavy reliance on mutation, i.e., SM and MM. This is indicative of the local-search nature of standard subtree crossover as documented in the early work of [31].

4.3.1 Factorial results

For the problem of Factorial, evolution was able to discover solutions using all different variation operator setups. Random search was also successful 10% of the times. MM yields 59% probability of success and outperforms all other search schemes (Table 6). SM is also very successful, attaining a 52% probability of success.

The learning curves of Figure 7(a), Figure 7(b), Figure 7(c), Figure 7(d) show that in all operator setups, except the UR+PM, evolutionary progress seems to stagnate at around generation 80. For UR+PM stagnation is realised around generation 30. SR+SM seems to learn quicker than MM and SM, and ultimately converges to lower error levels. We also observe that in all three SR+SM, SM, and MM most of the solutions are discovered prior to generation 20.

Figure 6(a) shows that the probability of success increases up to generation 70 for the mutation-based search regimes of SM and MM. The probability of success of recombination-based regimes SC+SM, UR+PM increases only up to generation 40. We attribute this difference to the quicker convergence that is observed in the learning curves of recombination-based regimes in Figure 7(c) and Figure 7(d).

The evolution of error entropy in Figure 8(a), Figure 8(b), Figure 8(c), Figure 8(d) show that on average the phenotypic diversity increases as a function of time for the mutation-based cases of SM and MM. For the recombination-based search regimes phenotypic diversity is either constant for UR+PM, or decreasing for SR+SM as a function of time.

4.3.2 Fibonacci results

The problem of Fibonacci turned out to be difficult for GP. UR+PM was unable to discover any solutions; random search was similarly unsuccessful. MM was the best performing, discovering a solution in 3 out of 100 runs. Learning curves in Figure 7(e) and Figure 7(h) suggest that evolution stagnates at around generation 80 and 60 for the cases of SR+SM and UR+PM respectively. No particular stagnation is demonstrated in the case of MM. Similarly to the problem of Factorial, SR+SM learns faster and ultimately converges to lower error levels as opposed to other search regimes.

Figure 6(b) shows that in all search schemes, the probability of success increases up to generation 58, after which no new solutions are ever discovered. On average, the evolution of phenotypic diversity in Figure 8(g) shows that diversity is an increasing function of time for the case of MM. After generation 20, the diversity in SM and UR+PM seem to be constant on average as a function of time in Figure 8(f) and Figure 8(h) respectively. The decrease in phenotypic diversity is rapid (after approx. generation 15) for the case of SR+SM. This is indicative of a local search not ideal to explore the search space of recursive programs effectively.

4.3.3 Exponentiation results

Similarly to Fibonacci, this is also a difficult problem for GP to solve. We suggest that this difficulty partly stems from the fact that both problems of Fibonacci and Exponentiation have two parameters that need to be combined in the evolved solution as opposed to the problem of Factorial with a single input parameter. All variation operator schemes were able to discover solutions; MM performed the best with a probability of success of 5%, while random search was successful in 2% of runs.

Learning curves in Figure 7 show no substantial convergence of error but for the case of UR+PM. Similarly to previous problems, SR+SM learns faster and converges to lower error. The error entropy histogram in Figure 8(k) shows that phenotypic diversity is an increasing function of time for the case of MM. In all search operator schemes, diversity appears to remain constant throughout the evolutionary runs after a time-dependent threshold.

4.3.4 Even-n-parity results

MM outperformed all other operator setups, and attained a probability of success of 21%. UR+PM was unable to discover solutions for this problem. The learning curves for Even-*n*-parity in Figure 7 show no particular convergence for the mutation-based operators. SR+SM seems to converge after generation 60, while UR+PM performs badly by getting stuck in local optima at the very early stages of evolution (after approx. generation 8). In addition, SR+SM learns faster and converges to lower error levels than the cases of mutation.

Figure 6(d) shows benefit from continuing the evolutionary runs past generation 75 for the case of MM; runs for SR+SM and SM found no additional solutions past that point. Error entropy curves in Figure 8 show that diversity is an increasing function of time for the case of MM; for all other search regimes diversity appears to decrease throughout the runs.

4.3.5 Nth results

In the problem of Nth, MM outperforms all other operators in terms of probability of success; random search discovered solutions in 2% of the runs. The learning curves in Figure 7 show that runs of MM and SR+SM converge on average around generation 80, while the runs of SM and UR+PM converge much faster.

Figure 6(e) shows that runs of MM benefit from an extended search past generation 62, while the rest of the operators are unable to discover any additional solutions past that point. Runs of SR+SM and SM appear to lose phenotypic diversity throughout the run in Figure 8(q) and in Figure 8(r) respectively. Phenotypic diversity appears to be an increasing function of time for the case of MM, and constant after a small time-dependent threshold for the case of UR+PM.

5 Lessons from the empirical analysis

The evolution of general recursive programs by means of GP is not a trivial exercise. This section considers different ways in which we can use the lessons learned from the empirical analysis to better search recursive program spaces. At the same time, we identify promising areas of future research.

5.1 Finess function

Our simulations suggested that given the current fitness function and subtree mutation, recursive programs are objects exhibiting the property of weak causality. The vast majority of fitness functions currently used in recursive GP heavily penalise non-halting programs. We believe that this may introduce a level of deception given that very small changes in the structure of a halting program can result in a non-halting program and therefore in a very bad individual. A non-halting program is not necessarily a bad program; it is possible that it contains useful parts that could be exploited throughout the course of evolution. Ideally, the fitness function needs to incrementally encourage the synthesis of a recursive solution by rewarding partially-correct recursive programs.

One possibility is to use a multi-objective fitness function that evaluates the semantics of a recursive program. Specific tests may be devised to check for the presence of recursive and base cases in an individual. As an example, in the case of Factorial a test may evaluate whether successive recursive calls are invoked on a decrementing argument. Notably, such a fitness function, similarly to the ones used in [6, 28] requires knowledge of the solution and lacks generality.

The fitness function we employed in this work is general and requires no knowledge about the evolved solution. Undoubtedly, this generality comes at the price of having to assign a pre-determined penalty to a non-halting program. In addition, a bias is exerted by the hard limit on the allowed number of recursive calls. We attempted to reduce this bias as much as possible by setting the number of allowed calls to $10,000$.

Furthermore, we recommend that the fitness function we used be combined with low selection pressure. This will allow for more exploration and somewhat counteract the tendency of the search to move away from non-halting programs solely on the ground that those programs exceeded the allowed recursive calls.

An alternative implementation of the current fitness function may use a less severe penalty for a non-halting program. Methods for credit/blame assignment [38] is a potential area for further research.

5.2 Variation operator

For evolutionary systems with heavy reliance on recombination, it is of great importance that the population of programs remains diverse. It turns out that fitness diversity can be maintained in standard symbolic regression applications of GP without the use of mutation. Nonetheless, if recursive nodes are prematurely lost from the population, evolutionary search will struggle in recursive program spaces. As it currently stands, non-halting programs are heavily penalised. Initial random programs containing recursive nodes are poorly structured, and are most likely non-terminating ones. It is difficult for these recursive programs to survive the selection process, and these get quickly eliminated from the population. Given the low probability of subtree mutation in SR+SM, the search concentrates around non-recursive programs. Search at that point becomes local and biased [31], as demonstrated in the fast convergence of learning curves to low error levels in Figure 7.

The very poor performance of UR+PM can be similarly attributed to premature elimination of recursive nodes from individuals in the population. However, in the total absence of subtree mutation this effect is even more pronounced, thus the operator was unable to discover any solutions for the problems of Fibonacci and Even-*n*-parity. We noticed that the type and arity constraints of primitive elements in all problems make it impossible for a non-recursive node to be mutated back to a recursive node using point-mutation. In the absence of subtree mutation, once programs containing recursive nodes wither away from the population, it is impossible to be introduced again. The search quickly converges to parts of the space with non-recursive programs, and gets stuck there indefinitely.

The interplay between the fitness function and the variation operator is crucial. Our simulations suggest that searching the space of recursive programs can be most effectively realised using a search operator scheme that relies heavily on mutation given our fitness function. Multi-mutation is able to increase the fitness diversity in the population. This turned out to be positively correlated with the likelihood of discovering solutions to recursive problems.

5.3 Handling the halting problem

To avoid the problem caused by non-terminating recursive programs we imposed a limit on the allowed number of intermediate recursive calls. By limiting the number of recursive calls in this way, we are providing a weak form of bias by constraining the range of different recursive behaviours that are considered during search. On the positive side, handling infinite recursion via a hard limit on recursive calls is straightforward to implement and to configure. In addition, it allows one to set a reasonable limit on the run-time of an experiment.

Is is possible to adapt this limit using a self-adaptive evolutionary computation [7]. In a self-adaptive evolutionary computation, adaptation of parameters can take place both on the population-level and on the individual-level. The limit of allowed recursive calls is a scalar that can be encoded along with an expression-tree of an individual, and then optimised. Furthermore, the recursive calls limit can be set my means of competitive coevolution, in which the difficulty of the training examples and therefore the respective computation budget are incrementally adapted as programs become fitter. An example of competitive coevolution between the difficulty of the task and the programs is reported for a reinforcement learning problem in [16].

Two interesting alternatives for tackling the halting problem, which have not been attempted in the context of evolving recursive programs, are the *competing coroutines* method of Maxwell [26], and the *anytime* method of Teller [37]. In the *competing subroutines* method, a population of programs is run concurrently, with best fitness being assigned to the first programs to provide correct output. In the *anytime* method, an evolved program is allowed access to a memory store, which can be manipulated through the use of special `read` and `write` primitives. Upon fitness evaluation, each program is given a short amount of time to run. Then the values of certain memory positions are extracted and interpreted as its answer regardless of whether it had terminated or not.

6 Conclusion

We presented an empirical analysis of the fitness landscape of recursive programs, and we subsequently evolved recursive solutions to a number of problems. Very importantly, the form of explicit recursion that we studied is general and in-line with conventional programming's implementation of recursive calls. It does not require high-level recursive operators to be supplied in the function set. At a minimum, the implementation requires a recursive function-node to represent recursive cases and a conditional `If-Then-Else` primitive to terminate recursion.

Simulations suggest that the functionality in parts of the space containing short programs converges to a limiting distribution after a small size-threshold. Overall, the proportion of halting programs decreases as the size of programs increases. We are expecting this finding to generalise in parts of the search space containing bigger-sized programs; this remains to be confirmed in future research.

In addition, landscape analysis showed that recursive programs are objects exhibiting the property of weak causality; small changes in the program structure may cause big changes in semantics and fitness. Nonetheless, the problem of evolving recursive programs is by no means a needle-in-a-haystack problem. There exist neighbours (in terms of small edit-distance) in the reach of a standard mutation-based operator that may be transitioned to in the course of hill-climbing towards an optimal program.

Given the fitness function employed, the multi-mutation operator is the most well-suited to evolve recursive solutions. This operator enabled high levels of phenotypic diversity in the population, and this correlated positively with the probability of discovering optimal programs. Evolution was also shown to outperform random search.

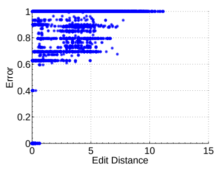
Recursion is an ingenious approach to problem solving by means of a computer program. Investigating scalable methods for evolving recursive programs expands the class of problems that GP can solve. Our work has direct implications for the broader project of general program synthesis by means of GP.

References

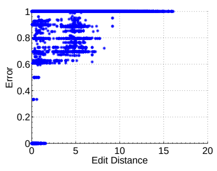
1. Agapitos, A., Lucas, S.M.: Evolving efficient recursive sorting algorithms. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, pp. 9227–9234. IEEE Press, Vancouver (2006)
2. Agapitos, A., Lucas, S.M.: Learning recursive functions with object oriented genetic programming. In: Proceedings of the 9th European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 3905, pp. 166–177. Springer, Budapest, Hungary (2006)
3. Agapitos, A., Lucas, S.M.: Evolving a statistics class using object oriented evolutionary programming. In: M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, A.I. Esparcia-Alcázar (eds.) Proceedings of the 10th European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 4445, pp. 291–300. Springer, Valencia, Spain (2007)
4. Agapitos, A., Lucas, S.M.: Evolving modular recursive sorting algorithms. In: M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, A.I. Esparcia-Alcázar (eds.) Proceedings of the 10th European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 4445, pp. 301–310. Springer, Valencia, Spain (2007)
5. Agapitos, A., McDermott, J., O'Neill, M., Kattan, A., Brabazon, A.: Higher order functions for kernel regression. In: M. Nicolau, K. Krawiec, M.I. Heywood, M. Castelli, P. Garcia-Sanchez, J.J. Merelo, V.M. Rivas Santos, K. Sim (eds.) 17th European Conference on Genetic Programming, *LNCS*, vol. 8599, pp. 1–12. Springer, Granada, Spain (2014)
6. Alexander, B., Zacher, B.: Boosting search for recursive functions using partial call-trees. In: T. Bartz-Beielstein, J. Brank, J. Smith (eds.) 13th International Conference on Parallel Problem Solving from Nature, *Lecture Notes in Computer Science*, vol. 8672, pp. 384–393. Springer, Ljubljana, Slovenia (2014)
7. Angeline, P.J.: Adaptive and self-adaptive evolutionary computations. In: Computational Intelligence: A Dynamic Systems Perspective, pp. 152–163. IEEE Press (1995)
8. Brave, S.: Evolving recursive programs for tree search. In: Advances in Genetic Programming 2. MIT Press (1996)
9. Burke, E.K., Gustafson, S., Kendall, G.: Diversity in genetic programming: An analysis of measures and correlation with fitness. IEEE Transactions on Evolutionary Computation **8**(1), 47–62 (2004)
10. Castle, T., Johnson, C.G.: Evolving high-level imperative program trees with strongly formed genetic programming. In: A. Moraglio, S. Silva, K. Krawiec, P. Machado, C. Cotta (eds.) Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012, *LNCS*, vol. 7244, pp. 1–12. Springer Verlag, Malaga, Spain (2012)
11. Castle, T., Johnson, C.G.: Evolving program trees with limited scope variable declarations. In: X. Li (ed.) Proceedings of the 2012 IEEE Congress on Evolutionary Computation, pp. 2250–2257. IEEE Computational Intelligence Society, IEEE Press, Brisbane, Australia (2012)

12. Chellapilla, K.: Evolving computer programs without subtree crossover. IEEE Transactions on Evolutionary Computation **1**(3), 209–216 (1997)
13. Clack, C., Yu, T.: Performance enhanced genetic programming. In: Proceedings of the Sixth Conference on Evolutionary Programming (1997)
14. Ekart, A., Nemeth, S.Z.: A metric for genetic programs and fitness sharing. In: R. Poli, W. Banzhaf, W.B. Langdon, J.F. Miller, P. Nordin, T.C. Fogarty (eds.) Genetic Programming, Proceedings of EuroGP'2000, *LNCS*, vol. 1802, pp. 259–270. Springer-Verlag, Edinburgh (2000)
15. Huelsbergen, L.: Learning recursive sequences via evolution of machine-language programs. In: Genetic Programming 1997: Proceedings of the Second Annual Conference
16. Kelly, S., Lichodzijewski, P., Heywood, M.I.: On run time libraries and hierarchical symbiosis. In: X. Li (ed.) Proceedings of the 2012 IEEE Congress on Evolutionary Computation, pp. 3278–3285. Brisbane, Australia (2012)
17. Kirshenbaum, E.: Iteration over vectors in genetic programming. Technical Report HPL-2001-327, HP Laboratories (2001)
18. Kochenderfer, M.J.: Evolving hierarchical and recursive teleo-reactive programs through genetic programming. In: Genetic Programming, Proceedings of EuroGP'2003 (2003)
19. Koza, J.: Genetic Programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA (1992)
20. Koza, J.: Genetic Programming II: automatic discovery of reusable programs. MIT Press, Cambridge, MA (1994)
21. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M.: Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufman
22. Langdon, W.B.: Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines **1**(1/2), 95–119 (2000)
23. Langdon, W.B.: Scaling of program functionality. Genetic Programming and Evolvable Machines **10**(1), 5–36 (2009)
24. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag (2002)
25. Langdon, W.B., Poli, R.: The halting probability in von Neumann architectures. In: P. Collet, M. Tomassini, M. Ebner, S. Gustafson, A. Ekárt (eds.) Proceedings of the 9th European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 3905, pp. 225–237. Springer, Budapest, Hungary (2006)
26. Maxwell, S.R.: Experiments with a coroutine execution model for genetic programming. In: IEEE Conference on Evolutionary Computation, pp. 413 – 417. IEEE (1994)
27. McDermott, J., Byrne, J., Swafford, J.M., O'Neill, M., Brabazon, A.: Higher-order functions in aesthetic EC encodings. In: 2010 IEEE World Congress on Computational Intelligence, pp. 2816–2823. IEEE Computation Intelligence Society, IEEE Press, Barcelona, Spain (2010)
28. Moraglio, A., Otero, F., Johnson, C., Thompson, S., Freitas, A.: Evolving recursive programs using non-recursive scaffolding. In: X. Li (ed.) Proceedings of the 2012 IEEE Congress on Evolutionary Computation, pp. 2242–2249 (2012)
29. Nishiguchi, M., Fujimoto, Y.: Evolutions of recursive programs with multi-niche genetic programming (mnGP). In: Proceedings of the 1998 IEEE World Congress on Computational Intelligence, pp. 247–252. IEEE Press, Anchorage, Alaska, USA (1998)
30. Nordin, P., Banzhaf, W.: Evolving turing-complete programs for a register machine with self-modifying code. In: L. Eshelman (ed.) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), pp. 318–325. Morgan Kaufmann, Pittsburgh, PA, USA (1995)
31. Poli, R., Langdon, W.B.: On the search properties of different crossover operators in genetic programming. In: J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, R. Riolo (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference, pp. 293–301. Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA (1998)
32. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008)
33. Rosca, J., Ballard, D.H.: Causality in genetic programming. In: L. Eshelman (ed.) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), pp. 256–263. Morgan Kaufmann, Pittsburgh, PA, USA (1995)
34. Rosca, J.P.: Entropy-driven adaptive representation. In: J.P. Rosca (ed.) Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, pp. 23–32. Tahoe City, California, USA (1995)
35. Shirakawa, S., Nagao, T.: Graph structured program evolution: Evolution of loop structures. In: R.L. Riolo, U.M. O'Reilly, T. McConaghy (eds.) Genetic Programming Theory and Practice VII, Genetic and Evolutionary Computation, chap. 11, pp. 177–194. Springer, Ann Arbor (2009)

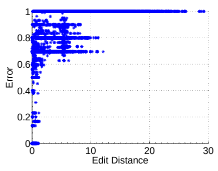
36. Spector, L., Klein, J., Keijzer, M.: The push3 execution stack and the evolution of control. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1689–1696 (2005)
37. Teller, A.: Genetic programming, indexed memory, the halting problem, and other curiosities. In: Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium, pp. 270–274. IEEE Press, Pensacola, Florida, USA (1994)
38. Teller, A., Veloso, M.: Efficient learning through evolution: Neural programming and internal reinforcement. In: Proceedings of the Seventeenth International Conference on Machine Learning (2000)
39. Turner, A., Miller, J.: Recurrent cartesian genetic programming. In: T. Bartz-Beielstein, J. Branke, B. Filipic, J. Smith (eds.) 13th International Conference on Parallel Problem Solving from Nature, *Lecture Notes in Computer Science*, vol. 8672, pp. 476–486. Springer, Ljubljana, Slovenia (2014)
40. Whigham, P.A., McKay, R.I.: Genetic approaches to learning recursive relations. In: X. Yao (ed.) Progress in Evolutionary Computation, *Lecture Notes in Artificial Intelligence*, vol. 956, pp. 17–27. Springer-Verlag (1995)
41. Wilson, G., Heywood, M.: Learning recursive programs with cooperative coevolution of genetic code mapping and genotype. In: D. Thierens, H.G. Beyer, J. Bongard, J. Branke, J.A. Clark, D. Cliff, C.B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J.F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K.O. Stanley, T. Stutzle, R.A. Watson, I. Wegener (eds.) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, vol. 1, pp. 1053–1061. London (2007)
42. Wong, M.L.: Evolving recursive programs by using adaptive grammar based genetic programming. Genetic Programming and Evolvable Machines **6**(4), 421–455 (2005)
43. Wong, M.L., Leung, K.S.: Evolving recursive functions for the even-parity problem using genetic programming. In: Advances in Genetic Programming 2. MIT Press (1996)
44. Wong, M.L., Leung, K.S.: Learning recursive functions from noisy examples using generic genetic programming. In: J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo (eds.) Genetic Programming 1996: Proceedings of the First Annual Conference, pp. 238–246. MIT Press, Stanford University, CA, USA (1996)
45. Yu, T.: Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. Genetic Programming and Evolvable Machines **2**(4), 345–380 (2001)
46. Yu, T.: A higher-order function approach to evolve recursive programs. In: T. Yu, R.L. Riolo, B. Worzel (eds.) Genetic Programming Theory and Practice III, *Genetic Programming*, vol. 9, chap. 7, pp. 93–108. Springer, Ann Arbor (2005)
47. Yu, T., Clack, C.: Recursion, lambda abstractions and genetic programming. In: J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, R. Riolo (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference, pp. 422–431. Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA



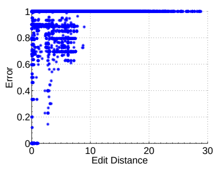
(a) Factorial
(Size: 15, Depth: 4)



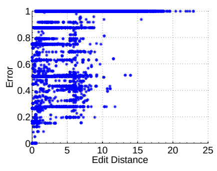
(b) Factorial
(Size: 47, Depth: 7)



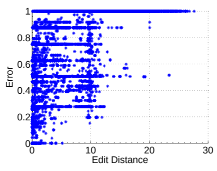
(c) Factorial
(Size: 83, Depth: 10)



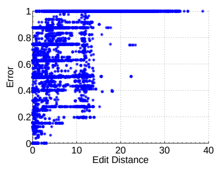
(d) Factorial
(Size: 183, Depth: 10)



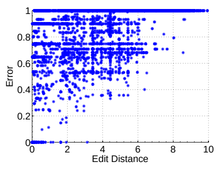
(e) Fibonacci
(Size: 31, Depth: 6)



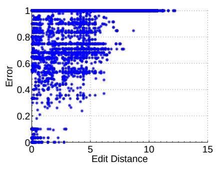
(f) Fibonacci
(Size: 70, Depth: 7)



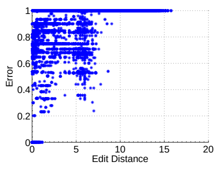
(g) Fibonacci
(Size: 73, Depth: 10)



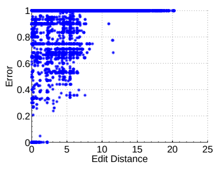
(h) Exponentiation
(Size: 29, Depth: 7)



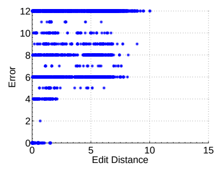
(i) Exponentiation
(Size: 31, Depth: 6)



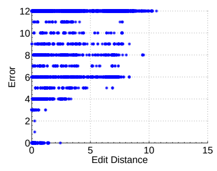
(j) Exponentiation
(Size: 67, Depth: 8)



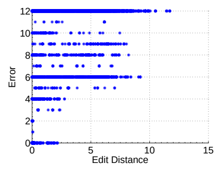
(k) Exponentiation
(Size: 73, Depth: 10)



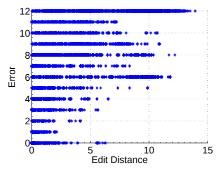
(l) Even-n-Parity
(Size: 29, Depth: 7)



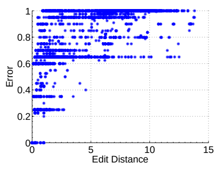
(m) Even-n-Parity
(Size: 63, Depth: 9)



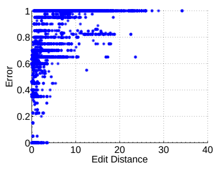
(n) Even-n-Parity
(Size: 112, Depth: 10)



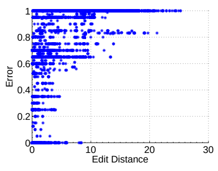
(o) Even-n-Parity
(Size: 447, Depth: 10)



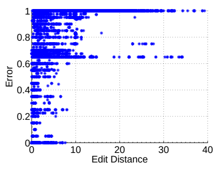
(p) Nth_
(Size: 19, Depth: 4)



(q) Nth_
(Size: 44, Depth: 8)

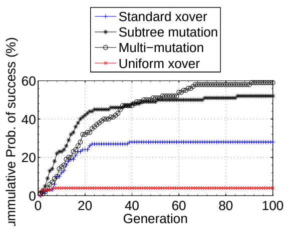


(r) Nth_
(Size: 74, Depth: 7)

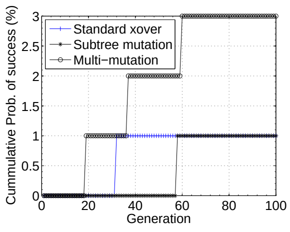


(s) Nth_
(Size: 222, Depth:10)

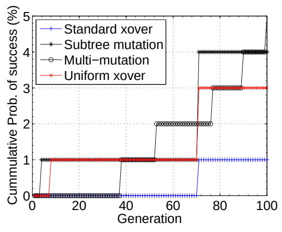
Fig. 5 Scatter plots between edit-distance and error. Error refers to the value of the loss function. For each problem we used evolved solutions of ascending expression-tree size highlighted in grey in Table 5.



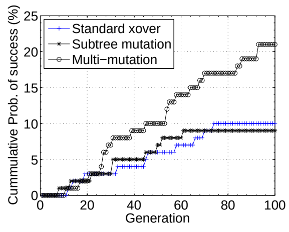
(a) Factorial



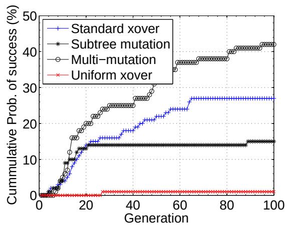
(b) Fibonacci



(c) Exponentiation

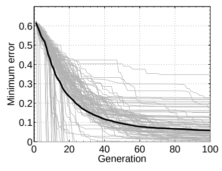


(d) Even-n-Parity

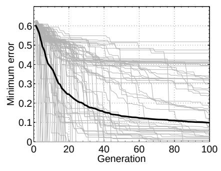


(e) Nth

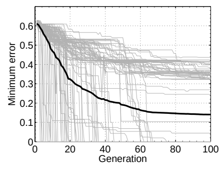
Fig. 6 Cumulative probability of success.



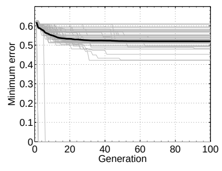
(a) Factorial (SR+SM)



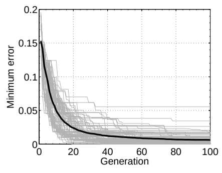
(b) Factorial (SM)



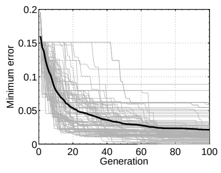
(c) Factorial (MM)



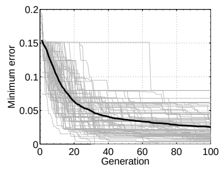
(d) Factorial (UR+PM)



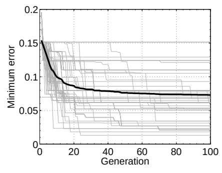
(e) Fibonacci (SR+SM)



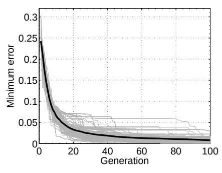
(f) Fibonacci (SM)



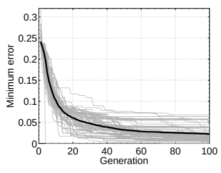
(g) Fibonacci (MM)



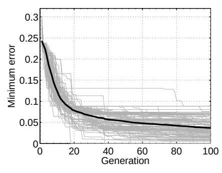
(h) Fibonacci (UR+PM)



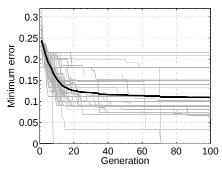
(i) Exponent. (SR+SM)



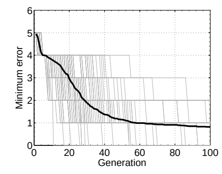
(j) Exponent. (SM)



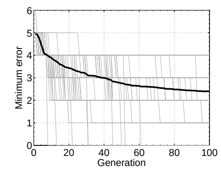
(k) Exponent. (MM)



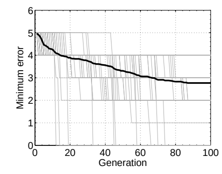
(l) Exponent. (UR+PM)



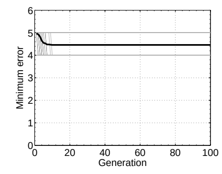
(m) Even-n-Par (SR+SM)



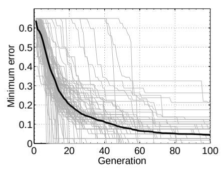
(n) Even-n-Par (SM)



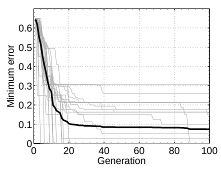
(o) Even-n-Par (MM)



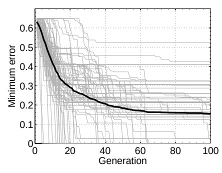
(p) Even-n-Par (UR+PM)



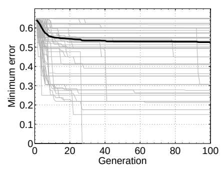
(q) Nth (SR+SM)



(r) Nth (SM)

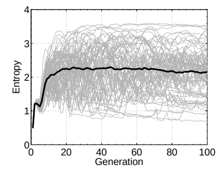


(s) Nth (MM)

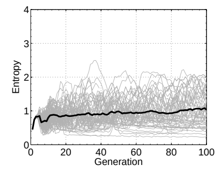


(t) Nth (UR+MM)

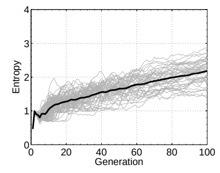
Fig. 7 Evolution of best-of-generation error (minimum error) in 100 independent evolutionary runs using different variation operators. Mean shown in bold.



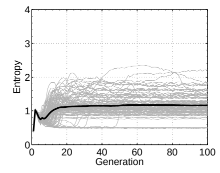
(a) Factorial (SR+SM)



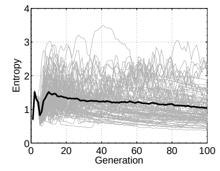
(b) Factorial (SM)



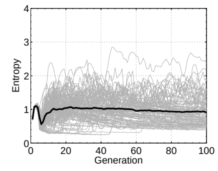
(c) Factorial (MM)



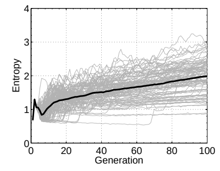
(d) Factorial (UR+PM)



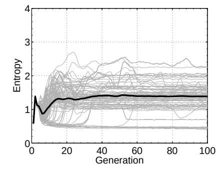
(e) Fibonacci (SR+SM)



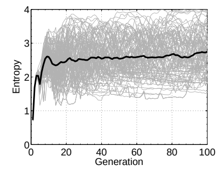
(f) Fibonacci (SM)



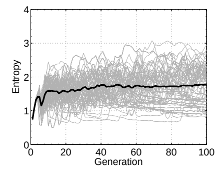
(g) Fibonacci (MM)



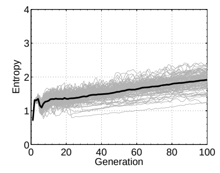
(h) Fibonacci (UR+PM)



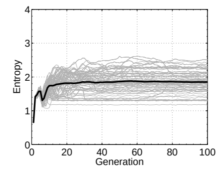
(i) Exponent. (SR+SM)



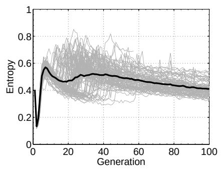
(j) Exponent. (SM)



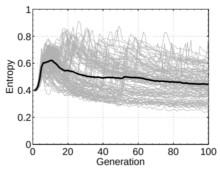
(k) Exponent. (MM)



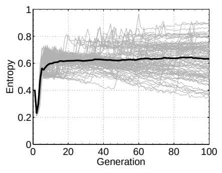
(l) Exponent. (UR+PM)



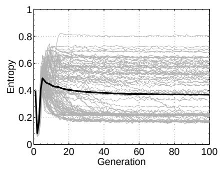
(m) Even-n-Par (SR+SM)



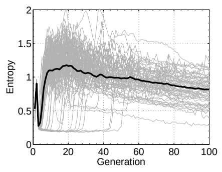
(n) Even-n-Par (SM)



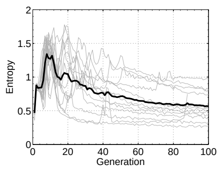
(o) Even-n-Par (MM)



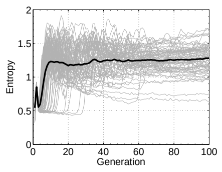
(p) Even-n-Par (UR+PM)



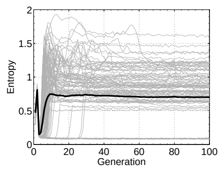
(q) Nth (SR+SM)



(r) Nth (SM)



(s) Nth (MM)



(t) Nth (UR+PM)

Fig. 8 Evolution of error entropy in 100 independent evolutionary runs using different variation operators. Mean shown in bold.