Geometric Semantic Genetic Programming for Recursive Boolean Programs

Alberto Moraglio
University of Exeter
Exeter, UK
A.Moraglio@exeter.ac.uk

Krzysztof Krawiec
Poznan University of Technology
Poznan, Poland
krawiec@cs.put.poznan.pl

ABSTRACT
Geometric Semantic Genetic Programming (GSGP) induces a unimodal fitness landscape for any problem that consists in finding a function fitting given input/output examples. Most of the work around GSGP to date has focused on real-world applications and on improving the originally proposed search operators, rather than on broadening its theoretical framework to new domains. We extend GSGP to recursive programs, a notoriously challenging domain with highly discontinuous fitness landscapes. We focus on programs that map variable-length Boolean lists to Boolean values, and design search operators that are provably efficient in the training phase and attain perfect generalization. Computational experiments complement the theory and demonstrate the superiority of the new operators to the conventional ones. This work provides new insights into the relations between program syntax and semantics, search operators and fitness landscapes, also for more general recursive domains.

CCS CONCEPTS
•Software and its engineering → Genetic programming; •Theory of computation → Evolutionary algorithms;

KEYWORDS
Genetic Programming, Semantics, Geometric Semantic Genetic Programming, Principled Design, Boolean Functions, Recursive Programs

ACM Reference format:
DOI: 10.1145/3071178.3071266

1 INTRODUCTION
For about a decade, Genetic Programming (GP) witnessed a trend towards a deeper understanding of program semantics and its effect on search [2, 6, 7, 12], which often led to improved forms of GP. GSGP [8] is a form of semantic GP with a strong theoretical foundation which induces a simple unimodal fitness landscape for any supervised machine learning problem and has provably good search performance [9].

Most of the work around GSGP to date has focused on real-world applications [15] and on improving the search operators for the originally proposed domains (Arithmetic, Boolean, Classifiers). Relatively little attention has been paid to using the theoretical framework behind GSGP for the principled design of search operators for other domains.

Recursion is a key construct in computer programs. There have been several attempts to evolve recursive programs [1, 3, 5, 11, 16, 18]. However, GP has been found badly suited to this [17]. A major challenge is that GP operators are highly disruptive when applied to recursive programs, because small changes in the code of a recursive program cascade through the recursion, amplifying the difference in behaviour. As a result, fitness values between parent and offspring programs may vary immensely, giving rise to highly discontinuous fitness landscapes.

In this paper, we embrace the challenge of designing search operators that provably see a unimodal landscape when evolving recursive programs. Our aim is to provide theoretical insights about how to design good semantic-aware search operators for recursive domains, by studying a small and well-defined domain – a kind of ‘onemax’ of recursive programs. We extend the GSGP framework to the domain of recursive programs that map variable-length Boolean lists to Boolean values. For this domain, we use the theoretical observations to guide the design of search operators that are provably efficient in the training phase and produce provably correct programs, attaining so perfect generalization.

2 GEOMETRIC OPERATORS FOR RECURSION
In this section, we design geometric semantic search operators for recursive Boolean problems, so that GP with these operators will be guaranteed to be efficient in the training phase.

2.1 Naive approach
The conventional geometric semantic search operators for the Boolean domain introduced in [8] are:

- Crossover: \( T_3 = (T_1 \land TR) \lor (\neg TR \land T_2) \), where \( TR \) is a random program.
- Mutation: \( TM = T \lor M(I) \) with probability 0.5 and \( TM = T \land \neg M \) with probability 0.5 where \( M(I) \) is a random minterm of all variables in program input \( I \).

When applied to Boolean programs of fixed input arity, i.e. \( \mathbb{B}^n \rightarrow \mathbb{B} \), these operators are geometric under the Hamming distance on the output vectors, i.e., mutation changes a single entry of the semantic vector, crossover produces offspring with intermediate semantics of parents, and the fitness landscape seen by a GP algorithm using these operators is unimodal with constant slope.
By extending the discourse to recursive programs, we move to the domain of programs with signature $\mathbb{BL} \to \mathbb{B}$ (BL domain for short), where $\mathbb{BL}$ is the space of all lists of Booleans. In this domain, it becomes convenient to rewrite the original operators using the if-then-else construct:

- Crossover: $T_1 = \text{if } TR \text{ then } T_1 \text{ else } T_2$
- Mutation: $TM = \text{if } M(l) \text{ then } RC \text{ else } T$, where $RC$ is a random constant true or false, each with prob $0.5$.

Note that these are the geometric semantic operators proposed in [8] for the domain of classifiers. This is not incidental, as classifiers form a super-domain of Boolean expressions, which can be thought of as binary classifiers fed with binary variables.

The random Boolean expression TR used originally in crossover naturally extends to a randomly generated (and possibly recursive) program $RP$ in the BL domain. For mutation, the original random minterm $M$ cannot be directly applied to program input, which is a variable-length list rather than a fixed-size set of named input variables. To make $M$ behave like a minterm for the BL domain, i.e. return true only for a single input and FALSE otherwise, we implement it as $eq(1, RL1st)$, where $RL1st$ is a random Boolean list of length smaller or equal than the maximum length of input vectors in the training set, and the primitive eq tests its arguments for equality. We then define the following operators:

- $SC(P_1,P_2) = \text{if } RP \text{ then } P_1 \text{ else } P_2$
- $SM(P) = \text{if } eq(I,RL1st) \text{ then } RC \text{ else } P$

$RP$, $RL1st$ and $RC$ are drawn uniformly and independently in each application of $SC$ and $SM$.

At this point it becomes essential to clarify how search operators affect recursion. To make sure that $SC$ and $SM$ preserve recursive calls in parent programs, we use generic call $\text{self}$, so that e.g. the factorial function can be expressed as $\text{fact}(n) = \text{if } n=0 \text{ then } 1 \text{ else } n \times \text{self}(n-1)$. The alternative approach of calling $\text{fact}$ explicitly would result in an offspring that calls its parents rather than itself.

### 2.2 Scaffolded GSGP operators

Though the above extension seems to naturally generalize GSGP to handle variable-length inputs, it is naive in ignoring the effects of recursive calls on program semantics. As a matter of fact, $SC$ and $SM$ do not induce a unimodal fitness landscape because the recursion interferes with the intended outcome of the functional application of GSGP search operators to recursive parents.

**Proof sketch:** Consider program of the form $p(i) = g(i, \text{self}(i-1))$, in which $\text{self}$, as per above convention, refers to $p$. Applying $SM$ to $p$ results in $o(i) = SM(g(i, \text{self}(i-1)))$, in which $\text{self}$ refers to $a$. This is not equivalent to the intended offspring $o(i) = SM(p(i))$ with $p(l)=SM(g(i, p(l-1)))$, as $SM$ is not inside the body of resulting recursive program. An analogous reasoning holds for semantic crossover.

This non-geometric character $SM$ and $SC$ can be seen as a special case of the more general brittleness of recursive programs, i.e. that a modification introduced by a search operator propagates through the stack of recursive calls and completely changes the behavior of a program (even though the change might have been intended to be minor). To address this brittleness in the general, non-semantic context, Moraglio et al [10] proposed scaffolding, a technique that substitutes the recursive calls with calls to the (unknown) target program, using the fitness cases as a surrogate for it. This allows evolving recursive programs as if they were non-recursive. Conceptually, scaffolding replaces the calls of the form $\text{self}(i-1)$ with $t(i-1)$, where $t$ is the target function, i.e. $t(i)$ returns the desired output for input $i$. $t$ can be seen a partial function that defines the desired behavior of program on the training examples only.

It should be clear at this point that we denote the argument of recursive call by $i-1$ mostly for clarity and conformance with the earlier example of factorial. The way in which that argument is formed depends on the specific domain. Crucially, scaffolding allows in recursive calls only arguments that are present in the training set (i.e. belong to the domain of the partial target function $t$). In other words, for a given input $i$ all its ‘predecessors’ that can be generated by the argument to recursive calls have to be present in the training set. This assumption implies certain consequences that we will come back to later.

With scaffolding, GSGP operators cause no interference with recursive calls, inducing so unimodal landscape and becoming efficient operators for training recursive programs.

**Claim 1:** Operators $SC$ and $SM$ are geometric under scaffolding.

**Proof sketch:** Consider program $p(i) = g(i, \text{self}(i-1))$. When scaffolding, $p$ becomes $p'(i) = g(i, t(i-1))$. Applying $SM$ to $p$ leads to $o'(i) = SM(g(i, t(i-1)))$ which is equivalent to $o'(i) = SM(p'(i))$. Therefore, $SM$ has the desired effect of modifying the output vector of $p$ at exactly one position. Analogous reasoning holds for $SC$. As a result, there is no interference with search operators, as search takes place in the ‘scaffolded function space’. Once the optimum function $h(i, t(t(i-1)))$ in the scaffolded space is found, it can be ‘descaffolded’ to obtain optimal recursive function $h(i)=h(i, h(i-1))$.

**Claim 2:** scaffolded semantic hill-climber finds the optimum (with zero training error) in $m \log(m)$ where $m$ is the size of the training set, for any target problem.

**Proof sketch:** The semantic mutation corresponds in the semantic space to a bit-flip mutation on a binary string of size $m$, i.e., the output vector on the function undergoing mutation. The hill-climber takes then $m \log(m)$ evaluations to hit the optimum (well-known runtime result [4]).

### 3 DOMAIN LANGUAGE AND TARGET CLASS

**Domain language:** There are many programming languages that can express programs with signature $\mathbb{BL} \to \mathbb{B}$. When designing the language for this study, we aimed at a minimal set of instructions that is expressive enough to (i) represent possibly many functions with that signature and (ii) represent the semantic operators concisely. Concerning (i), note that the semantics of $SM$ and $SC$ need to be implemented in the programming language of consideration, in order to provide for closure. We chose a minimal LISP instruction set augmented with elementary Boolean operators, the recursive call $\text{self}$, and the eq operator (Fig. 1). Defining and calling additional functions is not allowed, so the language is not Turing-complete, and not all $\mathbb{BL} \to \mathbb{B}$ functions are executable. Nevertheless the language allows implementing many useful and interesting functions, from generalized $2$-ary Boolean operators like and and or to more sophisticated concepts like testing whether the values in the input list form some repetitive pattern. The presence of two types requires strongly-typed GP systems, where programs created at random and programs
where we term which we term

To deal with errors, we devise a penalization approach that has

1

ite

const-list

Figure 1: The grammar defining the set of considered programs. const-list is a constant list of Booleans, I is the input list, self implements the recursive call, eq tests two lists for equality, and ite stands for if-then-else. The starting symbol of the grammar is B.

produced by search operators are guaranteed to be type-correct. Note that SM and SC applied to type-correct parent programs produce type-correct offspring programs, as a consequence of being semantically well-defined.

Target functions: We consider a family of target functions with tunable difficulty that allow expressing common recursive functions in the adopted programming language. They generalise the well-known fold function defined as:

\[
\text{fold} (I) = \text{ite} (\text{empty} (I), bc_0, \\
bc_2 (\text{head} (I), \text{self} (\text{tail} (I))))
\]

where \(bc_2\) is a 2-ary Boolean function and \(bc_0\) is a Boolean constant. List-wise and, list-wise or and list-wise parity belong to this class, which we term 1-fold functions. This class can be generalised to a \(k\)-fold class that includes all functions with recursion order up to \(k\). For instance, the blueprint for 2-fold functions is:

\[
\text{fold} (I) = \text{ite} (\text{empty} (I), bc_0, \\
\text{ite} (\text{eq} (I, [\text{false}]), bc_1, \\
\text{ite} (\text{eq} (I, [\text{true}]), bc_2, \\
b \text{false} (\text{true} (I), \text{self} (\text{tail} (I)), \\
\text{head} (\text{tail} (I)), \text{self} (\text{tail} (\text{tail} (I))))))
\]

where \(bc_4\) is a 4-ary Boolean function, \(bc_i\) are Boolean constants, and \([\text{false}]\) denotes a list containing one element \text{false}. Higher order of recursions are obtained by recursive calls with successive tails of the input list. The so defined order of recursion is a natural generalisation of the concept of the order of recursion in, e.g., recursive formulas. For instance, factorial has the order of recursion one because \(\text{fact} (n) = \text{fact} (n-1)\), while the Fibonacci function has order two because \(\text{fib} (n) = \text{fib} (n-1) + \text{fib} (n-2)\).

Handling run-time errors: In the BL domain, syntactical correctness of a program does not guarantee its error-free execution. To deal with errors, we devise a penalization approach that has a natural semantic interpretation and, importantly, does not alter the unimodality of the fitness landscape. When a program applies head or tail to an empty list, or applies self to a list that is not shorter than its argument \(I^1\), we assume that its output for that input is a designated special value \(err\). As a consequence, program semantics is a ternary vector of three symbols: true, false, and err. The definition of semantic distance remains unchanged, i.e. it is the Hamming distance, and so does fitness, i.e. Hamming distance of program’s semantics from the target semantics. Because the target semantics does not ever contain err, each error for a training example (test) results in a unit penalty. This error handling allows us to apply semantic operators also to erroneous programs.

---

\(^1\)Recursive call with an argument list that is not shorter than \(I\) does not need to lead to infinite recursion, but preventing such cases is essential for scaffolding.
enabled scaffolding, or simply performs recursive call in remaining configurations.

Table 1 presents the performance indicators of particular configurations obtained from 50 runs of each configuration on each benchmark for \( k \in \{1, 4\} \) and \( n \in \{1, 5\} \). A run is considered successful if it yields a program that produces correct outputs for all training examples. The number of evaluations, generalisation error, and graph size concern best-of-run programs and are averaged over successful runs only; the blank table cells mark configurations where no run succeeded. Evaluation presents the \( \log_{10} \) of the number of evaluations elapsed. Generalisation error is the percentage of lists of lengths in \([n + 1, n + 2]\) for which the program produced incorrect output (or \( \text{err} \)). Graph size is the number of unique nodes in a program, i.e. the size of a program tree when 'compressed' to a graph. Graph size is more appropriate than size of program tree, as GSGP operators produce programs that refer to (call) the same ancestor programs multiple times. It is also consistent with natural implementation in functional programming languages (used in our software framework), where programs are immutable and thus there is no need for cloning code pieces. Functional implementation forms a natural alternative to the efficient implementation introduced by Vanneschi et al [14].

5 ANALYSIS 1

In this section, we confront the experimental results with expectations grounded in the theory.

For the success rate, from the theory we expect: (i) semantic HC and with scaffolding and semantic EA with scaffolding to have 100% success rate as the landscape seen is unimodal and the gap on the number of evaluations is generously large; (ii) semantic HC without scaffolding to have success rate lower than 100% as the landscape seen is not guaranteed to be unimodal and may contain local optima; (iii) in general all configurations with scaffolding to attain higher success rates, as they see a smoother landscape; (iv) EA configurations to attain higher success rate than HC, as having multiple working solutions lessens the risk of getting stuck in local optima. The experiments completely confirm (i) and (iv). (ii) is also confirmed, though the landscape seen using the naive semantic operators without scaffolding is often unimodal. Concerning (iii), the presence of scaffolding does not seem to improve the success rate except for the case of the semantic HC. In the previous work [10], scaffolding was shown to be helpful on average on a large suite of problems, so this may be due to the specific class of problems studied here.

For the number of fitness evaluations to reach the optimum, from the theory we expect: (i) the runtime of semantic HC with scaffolding to grow slowly with \( n \) (more precisely \( m \log m \) where \( m \) is the number of fitness cases), and to not depend on \( k \); (ii) semantic HC with scaffolding to converge faster than EA with scaffolding, as on a unimodal landscape HC converges faster than population-based algorithms; (iii) configurations with semantic operators (SGP) to scale much better than those with traditional operators (GP), as the former see a unimodal landscape. Experiments confirm (i), (ii) and (iii). In particular, semantic HC with and without scaffolding is very quick and scales much better than the other algorithms.

For generalization error, from the theory we expect: (i) semantic HC to not generalise well, as SM is designed with the sole purpose of making training efficient and will tend to memorise the training set without making use of recursive calls; (ii) semantic EA to generalise better than semantic HC, as \( SC \) tends to introduce new recursive calls in the offspring; also, semantic EA should generalise better on larger training sets, as this favours offspring using recursive calls to fit the training set rather than memorising one entry at a time; (iii) operators using semantics and scaffolding to lead to worst generalisation than standard GP, because they have not been designed with generalization in mind. Experiments confirm all these expectations. In particular, HC does not generalize better for increasing \( n \), which confirms that it does memorise the training set. Note also how for sufficiently large \( n \) relative to \( k \), semantic EA always achieves perfect generalisation, which implies making proper use of recursive calls. In general, population-based algorithms perform similarly whether scaffolded or not, and whether semantic or not. Their generalisation error improves with larger \( n \) and deteriorates for larger \( k \). For HC, standard GP generalises better than semantic HC.

For the graph size of zero-error programs, from the theory we expect: (i) the size of the final solution to be linear in function of the number of fitness evaluations for semantic approaches; (ii) larger sizes for population-based semantic approaches as they use crossover; (iii) smaller program sizes for standard GP than for SGP as SGP solutions grow inherently steadily in size while traditional GP do not have such a systematic bias. Experiments (not shown) confirm these expectations. SGP and GP produce programs of generally comparable sizes, but traditional GP tends to suffer less from program growth.

6 DESIGN OF SEARCH OPERATORS FOR GOOD GENERALISATION

In this section, we design new genetic geometric search operators that not only scale provably well in convergence on the training set, but also guarantee such programs to generalise provably well on all (infinitely many) unseen inputs.

6.1 Requirements for Good Generalisation

PAC-learning [13] is a theoretical framework for deriving guarantees on generalisation for classes of Boolean functions. It cannot be directly applied to synthesis of correct recursive Boolean programs, as the generalisation sought there is perfect, rather than only probably approximately correct. However, PAC-learning brings an important lesson about generalisation in general: (i) provably good generalisation can be achieved on suitably small function classes, and with a suitably large training set; (ii) generalisation is a property of only the class of problem considered and the size (and distribution) of the training examples, and not of the training algorithm used. The only requirement on the training algorithm is to (efficiently) find a function within the problem class with zero-error on the training set.

By analogy to fixed-length program semantics for fixed-arity programs, let infinite semantics of program \( P \) be the (infinite) vector (sequence) of outputs produced by \( P \) for all input lists, ordered w.r.t. increasing length (and arbitrarily otherwise).

Claim: 1) For the class of \( k \)-fold functions for any given \( k \), a prefix of finite length \( m_k \) of infinite semantics is sufficient to uniquely identify any function in the class up to functions with the same infinite semantics. 2) Any function in this class that has zero error

on examples corresponding to prefix elements generalises perfectly on any unseen input. This result holds for any training algorithm. Proof sketch: 1) There is a finite number of functions belonging to a given k-fold class. Consider the set S of infinite semantics of all those programs. Each pair of them is either equal or different. For each different pair of semantics s1, s2 ∈ S consider the shortest prefix that discriminates them. Its length must be finite, otherwise s1 and s2 cannot be different. Now let m_k be the maximum length of such prefixes over each pair of semantics in S. By construction the prefix of length m_k discriminates any two semantics in S if two semantics in S have the same prefix of length m_k, they must be identical i.e., ∀s1, s2 ∈ S : s1[1:m_k] = s2[1:m_k] ⇒ s1 = s2. 2) Note that the unknown target function t also belongs to the k-fold class, so its infinite semantics s_t ∈ S. As per 1), if any k-fold function f with infinite semantics s_f coincides with s_t on the prefix of length m_k, they cannot differ on the remaining elements, and it must hold that s_f ≡ s_t, and so f must have zero generalisation error on all examples.

For prefix lengths l < m_k, we expect generalisation error to decrease with l, as longer prefixes are more selective. This may be useful if m_k is reasonably small. For the 1-fold class, we enumerated all output sequences and determined that m_1 = 15.

### 6.2 k-fold GSGP Operators

The practical upshot of the arguments brought in the previous section is that by (i) evaluating programs on a sufficiently large number of training cases and (ii) performing search in the space of k-fold functions only, the resulting program is guaranteed to generalize perfectly if it achieves zero error on the training set. The former is beyond our control in real-world settings, so we leave it open. The latter depends on the design of initialization and search operators. Therefore, in the following we refine the scaffolded GSGP search operators to make sure that the search takes place in the k-fold space only.

To begin with, we assume that the initial population contains only k-fold programs. This can be easily achieved using appropriate initialization operator, e.g. the procedure for generating k-fold target functions described in Section 4. Then, we propose the following two mutation operators:

\[ \text{SM}_1(k) = \text{ite(eq(I, r)), bc, P} \]
\[ \text{SM}_2(k) = \text{ite(len(I) ≥ k, SMB(P), P)} \]

where bc is a Boolean constant, and r1 is sampled from the set of all Boolean lists of length up to k − 1, and SMB is the semantic (minterm) mutation for Boolean expressions. We combine \( \text{SM}_1(k) \) and \( \text{SM}_2(k) \) into one k-fold mutation operator \( \text{SM}_k \) by using the former with probability \( n_c/(n_c + n_f) \) and the latter with probability \( n_f/(n_c + n_f) \), where \( n_c = 2^k - 1 \) and \( n_f = 2^k \). We define k-fold crossover as:

\[ \text{SC}_k(P_1, P_2) = \text{ite}(rf, P_1, P_2) \]

where \( rf \) is a random k-fold function.

In the following we show that \( \text{SM}_k \) and \( \text{SC}_k \) are k-fold preserving, i.e., given parents \( p, p' \in F_k \), the offspring \( \text{SM}_k(p), \text{SC}_k(p, p') \) are also in \( F_k \). We show this only for \( k = 1 \), but the reasoning holds for any \( k \). As shown in Section 3, functions in \( F_1 \) can be written in the 1-fold form as \( \text{fold}(I) = \text{ite( empty(I), bc, bf( head(I), self( tail(I) )))} \), where bc is a Boolean constant and bf is a 2-ary Boolean function. If a function \( g \) can be rewritten in this form, then
Some extracted text
s_1(g') (which is the same as s_1(g). For instance, when t is the function that always returns 0, we have: g' ([]) = g1 = f(0, 0) = 0, g' ([[]]) = g2 = f(0, 0) = 0, g' ([[]]) = g3 = f(1, 0) = 0, g' ([[]]) = g4 = f(0, 0) = 1. Moreover, the entries in s_k(g') can be grouped into disjoint blocks, each corresponding to one entry in s_i(g') and having the same output as that entry. For instance, g' ([[]]) and g' ([[]]) both belong to the block corresponding to the intensional semantic entry f(0, 0), and have the same S_1 output.

This implies the existence of a relation between the distance in the intensional semantic space (S_D) and the distance in the extensional semantic space (S_E). For any two k-fold functions f_1 and f_2, S_D(f_1, f_2) = HD(s_i(f_1), s_i(f_2)), where HD is Hamming distance. In turn, S_E(f_1, f_2) = HD(s_e(f_1), s_e(f_2)), but crucially S_D(f_1, f_2) can be also expressed as a weighted HD of their intensional semantics, in which the weight of each contributing entry is the size of the corresponding block in the extensional semantics.

For the fitness landscape seen from the intensional space it holds that for every point p at a distance S_D(p, t) to the target t, its fitness is Fit(p) = SE_e(s_e(P), t) = HD_w(s_e(P), s_e(t)). This implies that the fitness landscape as seen by the search operators in the intensional semantic space is still unimodal, as moving towards the target in the intensional space (smaller S_D(P, t)) corresponds to getting a better fitness (smaller HD_w(s_e(P), s_e(t))). However, it does not have a constant gradient (which is the case for the traditional GSGP operators). Nevertheless, when using an ordinal selection scheme (i.e., such that depends only on the order of fitness values, like tournament selection), this landscape is effectively equivalent to a unimodal landscape with a constant gradient.

7 EXPERIMENT 2

In this experiment we evaluate the k-fold search operators SM_k and SC_k described in Section 5.2. To implement them, we extend the set of instructions from Fig. 1 with rf_k and the terminal instruction len_k that implements the condition len(1) ≥ k, required in SM_k. We also assume this time that the constant lists (the const-1st terminal) are all Boolean lists shorter than k. This extended instruction set is sufficient to implement SM_k and SC_k, including the minterm operation SMB. These search operators form the configuration SGP rf_k.

Note that, despite featuring additional instructions, the set of programs considered here is not a superset of the language defined by the grammar in Fig. 1, because instructions can be combined only as prescribed by SM_k and SC_k, and so keep search within the bounds of rf_k. Also, recursive calls can be introduced only by mutation (contrary to SGP, where only crossover was capable of that), more precisely by the SMB term. As a consequence, new recursive calls may appear also in SGP run that use only mutation.

The baseline method is RS_k, the random search in the space of k-fold functions F_k. RS_k is a single-point hill climber that in each iteration generates a random k-fold program p = rf_k, and replaces the current program with p if the fitness of p is better.

With rf_k terminals present in instruction set, it becomes possible to find a single-node solution to a given k-fold target function benchmark. However, the odds for this decrease rapidly with k, as there are 2^{k-1} k-fold functions.

The remaining settings are identical to those in Experiment 1. The results of this experiment are shown in Table 2.

8 ANALYSIS 2

For success rate, hitting the optimum gets harder with growing k and growing n, which is particularly evident for RS. This is expected because the number of functions in a k-fold class grows exponentially with k, and the number of functions in F_k that match the training examples decreases with n (up to a critical number of fitness cases).

For the number of evaluations, the expected number of evaluations to hit the optimum in RS is the size of F_k divided by the number of functions in F_k that match all training examples. This ratio is also a measure of difficulty of the problem. Experimental results confirm that for small n and k the assumed population size of 1000 is disproportionate w.r.t. the expected performance of RS, and the optimum is found in the initial population. The results confirm also that HC (scaffolded or not) scales much better than RS for growing k and n, and that for n and k large enough EA also scales better than RS.

For generalisation performance, for 1-fold class, for n ≥ 3 all search algorithms generalize perfectly, as predicted by the theory. Perfect generalisation is independent from the search algorithm, and depends only on the problem class and sufficient number of fitness cases. When the training set is not large enough to guarantee perfect generalisation, all algorithms improve their generalisation performance for growing training set size as expected, and we notice that random search generalises the best.

For graph size (results not shown) the solutions found by random search are always the smallest, which must be the case all candidate solutions in RS_k are single-node k-fold functions rf_k. The sizes of the programs found by the other algorithms are smaller than those not using the k-fold operators.
9 CONCLUSIONS AND FUTURE WORK

Evolving recursive programs effectively is one of the long-standing challenges for GP and the key to its ultimate goal of evolving complex, fully-fledged programs from examples. In this paper, we made first steps towards designing provably efficient search operators for recursive domains that see a unimodal fitness landscape and are guaranteed to find the correct program. These three properties have significant implications for theory and practice of GP, and make this new line of investigation very exciting. The theoretical concepts and methodology introduced here for the specific case of recursive Boolean programs have in principle much wider applicability. There are several interesting lines of future investigation: extending this framework to arithmetic recursive domains and investigate the potential issues resulting from moving to continuous domains; extension to functions that map multiple Boolean lists to Boolean vectors of lists); investigating classes of functions that capture traditional LISP programs manipulating nested lists of symbols (i.e., tree-like structures), and so involve richer data structures that are inherently recursive; last but not least, extending the framework to Turing-complete programs.

Acknowledgment K. Krawiec acknowledges support from grant 2014/15/B/ST6/05205 funded by the National Science Centre, Poland.

REFERENCES