Chapter 1
Behavioral Program Synthesis: Insights and Prospects

Krzysztof Krawiec, Jerry Swan, and Una-May O'Reilly

Abstract Genetic programming (GP) is a stochastic, iterative generate-and-test approach to synthesizing programs from tests, i.e. examples of the desired input-output mapping. The number of passed tests, or the total error in continuous domains, is a natural objective measure of a program's performance and a common yardstick when experimentally comparing algorithms. In GP, it is also by default used to *guide* the evolutionary search process. An assumption that an objective function should also be an efficient 'search driver' is common for all metaheuristics, such as the evolutionary algorithms which GP is a member of. Programs are complex combinatorial structures that exhibit even more complex input-output behavior, and in this chapter we discuss why this complexity cannot be effectively reflected by a single scalar objective. In consequence, GP algorithms are systemically 'underinformed' about the characteristics of programs they operate on, and pay for this with unsatisfactory performance and limited scalability. This chapter advocates *behavioral program synthesis*, where programs are characterized by informative execution traces that enable multifaceted evaluation and substantially change the roles of components in an evolutionary infrastructure. We provide a unified perspective on past work in this area, discuss the consequences of the behavioral viewpoint, outlining the future avenues for program synthesis and the wider application areas that lie beyond.

Key words: genetic programming, program behavior, program semantics, multiobjective evaluation, search driver, evaluation bottleneck

Krzysztof Krawiec
Institute of Computing Science, Poznan University of Technology, Poznań, Poland

Jerry Swan
Department of Computer Science, University of York, UK

Una-May O'Reilly
ALFA, CSAIL, MIT, Cambridge, MA, USA

1.1 Introduction and motivations

Program synthesis is a challenging task due to the size of the search space, its multimodality, externalized semantics of instructions, and complex contextual interactions between them. These characteristics are intrinsic to the nature of this task and cannot be evaded. However, some difficulties faced by contemporary genetic programming (GP), in particular the far from satisfactory scalability, result from the particular model of evaluation of candidate solutions adopted in this generative, trial-and-error metaheuristic.

As in the majority of genres of evolutionary computation (EC), the candidate solutions (programs) in GP are conventionally evaluated using scalar, generic performance measures. Such a measure will usually capture program error, e.g. represented either as the number of failed tests (for discrete domains) or the total error committed on them (for the continuous domains).

The practice of measuring the quality of candidate programs using a scalar performance measure has several merits. It allows for strict and elegant formulation of program synthesis task as an optimization problem, and is thus compatible with the conventional way of posing problems in artificial intelligence, operational research, and machine learning. It also eases the separation of generic search algorithms from a domain-specific evaluation function, which is so vital for *meta*heuristics. No wonder that this 'design pattern' is so common that we rarely ponder its other consequences.

Unfortunately, there is a price to pay for all these conveniences, which arises from the inevitable loss of information that accompanies the process of scalar evaluation. That loss is particularly high in generate-and-test program synthesis like GP, where not only a program itself is a complex combinatorial entity, but also its execution is an intricate iterative process. In consequence, the spectrum of possible *behaviors* exhibited by programs is enormously rich. For example, even when looking only at program output, the number of all possible behaviors of programs that attempt to solve the (trivial for contemporary GP) problem of 6-bit multiplexer is the staggering 2^{64}. Yet, in conventional GP all that is left of that process is a single number (in the interval $[0, 64]$ for the above example). The conventional scalar evaluation *denies a search algorithm access to the more detailed information on program's behavioral characteristics*, while *that information could help to drive the search process more efficiently.*

This observation can be alternatively phrased using the message-passing metaphor typical in information theory. A search algorithm and an evaluation function can be likened to two parties that exchange messages. The message the algorithm sends to the evaluation function encodes the candidate solution to be evaluated. In response, the algorithm receives a return message – the evaluation. In a sense, the evaluation function *compresses* a candidate solution into its evaluation. If one insists on compressing all the information about program behavior into a scalar fitness that aggregates various aspects of that behavior, then one also has to accept the fact that such compression is inevitably lossy.

This *evaluation bottleneck* has detrimental consequences. The outcomes on particular tests compensate each other and may render programs indistinguishable in a selection phase, leading to loss of diversity and premature convergence. Also, tests may vary with respect to objective difficulty (the probability of a random program passing a test), subjective difficulty (measured by search algorithm's likelihood to find a program that passes the test), or both. In consequence, evolution often tends to greedily synthesize programs that pass the easiest tests, and such programs may correspond to local minima in the search space. These and other properties of conventional evaluation cause it to exhibit low fitness-distance correlation (Tomassini et al, 2005), i.e. to not reflect well the number of search steps required to reach the optimal solution. As a result, guiding search by a fitness function defined in this way may be not particularly efficient. In other words, the fitness function, despite embodying the objective quality of candidate solutions (considered as prospective outcomes of program synthesis process), is not necessarily the best driver to guide the search.

The parsimony of conventional evaluation is also awkward in architectural terms, i.e. when looking at a program synthesis system as a network of interconnected components. Why would one component (fitness function) compress the evaluation outcomes and then force another component (search algorithm) to reverse-engineer them, knowing that this incurs loss of information? There are no reasons for this other than the convention inherited from metaheuristic optimization algorithms and evolutionary metaphor.

Arguably, there are domains where an evaluation function is by definition 'opaque' and makes this bottleneck inevitable. For instance, in Black Box Optimization, fitness is the only information on a candidate solution available to a search algorithm. However, it might be the case that the need of such separation is more an exception than a rule when considering the whole gamut of problems we tackle with metaheuristics. In many domains, there are no principal reasons to conceal the details of evaluation, which is often complex and an abundant source of potentially useful information. This is particularly true for program synthesis, where the act of evaluating a candidate program is rich at least in two respects. Firstly, a program interacts with *multiple tests*, and will often perform differently on each. Secondly, a program's confrontation with a single test involves executing *multiple instructions*.

The main motivation for this chapter is the observation that the habit of driving search using a conventional, scalar evaluation function cripples the performance of stochastic program synthesis as implemented by GP. In response, we posit the necessity of broadening the evaluation bottleneck and providing search algorithms with more detailed information on program behavior. This leads to a new paradigm of *behavioral program synthesis*. In this chapter, we demonstrate a particular means to this end, presented earlier in preliminary forms in (Krawiec and Swan, 2013) and (Krawiec and O'Reilly, 2014), which relies on the concept of information-rich *search drivers*, alternative quasi-objectives that may be capable of guiding program synthesis process more efficiently than the conventional objective function.

1.2 Behavioral program synthesis

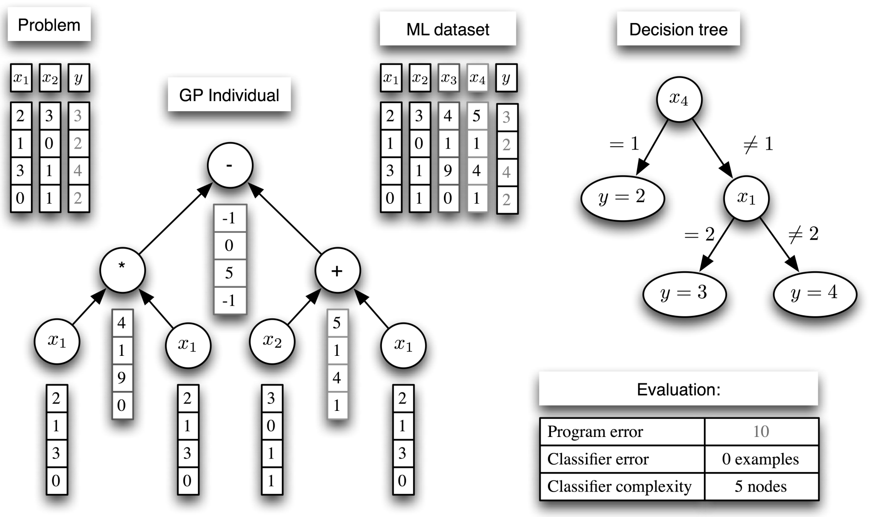
In this section we sketch the vision of *behavioral program synthesis*, a methodology for program synthesis that prioritizes program behavior.

Several existing extensions of the traditional GP paradigm involve broadening of evaluation bottleneck, in a more or less explicit way. For instance, program semantics in GP is the vector of program outputs for particular tests (Moraglio et al, 2012) and thus provides more information about program behavior than the conventional scalar fitness. Behavioral characterizations like program semantics are tailored to the needs of a particular approach: a semantics of a program holds program output for every test, because this is the information required by (most) semantic-aware search operators.

Contrary to this model, we propose that evaluation provides a complete account of program behavior, and to leave it up to the other components of a search algorithm to decide which pieces of that information to use. The means for this are *program trace*, which reports the detailed, instruction-by-instruction effects of program execution for a given input, and *execution record* that gathers and aligns such traces for all considered tests.

Both these concepts can be conveniently explained with an example. Figure 1.1 presents an integer-valued symbolic regression task ('Problem') defined by four tests, each of them comprising two input variables x_1 and x_2 and the desired output y. Assume the tree-based GP program p shown there ('GP Individual') needs to be evaluated. The colored lists present the outcomes of intermediate execution stages, produced by p at particular instructions for consecutive tests. When gathered together, they form the execution record (labeled 'ML dataset' in Fig. 1.1, for the reasons explained later). A single row in an execution record captures the behavior of p on the corresponding test in the set of tests; for instance, the first row does so for $x_1 = 2$ and $x_2 = 3$. For this input, the intermediate values generated by p at consecutive instructions are $2, 2, 3, 2, 4, 5$, when executing p in the bottom-top, left-to-right manner (the ordering could be different for this side effect-free programming language). The corresponding first row of the execution record presents this in an abridged form, where the duplicates are omitted: $2, 3, 4, 5$ (the second and the fourth leaf in the tree refer to the input variable x_1 that has been already recorded in the first element of the trace).

A trace is thus a sequence of intermediate computation states, and can be harvested from a running program by interrupting its execution after every instruction, and taking a 'snapshot' of the *execution environment.* In the above example with functional tree-based GP, a state is simply the working



Program error	10
Classifier error	0 examples
Classifier complexity	5 nodes

Fig. 1.1 The workflow of behavioral evaluation in Pattern-guided Program Synthesis (PANGEA, (Krawiec and Swan, 2013)), valid also for behavioral programming presented in (Krawiec and O'Reilly, 2014).

value returned by the currently executed node of an expression tree. Other representations used in GP require different implementations of this concept. In linear GP (Brameier and Banzhaf, 2007), statements operate via side-effects i.e. by changing the values stored in registers; the environment there would be the states of all registers. In the PushGP system (Spector and Robinson, 2002), where the working memory is the code stacks and data stacks, all these data structures taken together form the execution environment. Nevertheless, in both these cases recording traces is straightforward, as demonstrated by our use of PushGP in (Krawiec and Swan, 2013).

Differences between program representations notwithstanding, an execution record captures the entirety of effects of program's interactions with the input data provided in tests. As such, it is obviously possible to derive from it the conventional fitness (by comparing the last column with the vector of desired outputs), the outcomes of interactions with individual tests (which opens the door to posing a program synthesis task as a test-based problem (Popovici et al, 2011)), or program semantics (in the sense of semantic GP (Moraglio et al, 2012)). The arguably most exciting possibility (which has been little explored to date) lies in investigating 'internal' program behavior, which we attempt in the following.

For expressions like the one in the above example, the execution record is by definition *aligned*, i.e. the states recorded in the same column correspond to the same instruction in the program. For programs containing loops, conditional statements, or involving recursion, traces may have different length and alignment is not guaranteed. Nevertheless, this does not invalidate our hypothesis that certain regularities present in an execution record can be valuable telltales of program's actual or prospective performance. The particular approach presented in the next section exemplifies this claim.

1.3 Pattern-guided program synthesis

In conventional GP (and other conceivable generate-and-test program synthesis techniques), candidate programs are normally judged by their outputs. However in GP, arguably more than in many other domains, the ultimate program output is an effect of collective effort of constituent instructions. One reason for this state of affairs is the sequential nature of programs. The other is the particularly complex mapping from program code to behavior: a minute modification of the former may cause a dramatic change in the latter. On the other hand, a major change in a program can turn out to be behaviorally neutral, due to the multimodality mentioned above.

It is thus likely that programs emerge in an evolving population that feature potentially useful components (subprograms, code fragments) yet that usefulness is not leveraged by the final instructions. Such programs will usually perform poorly in terms of conventional fitness and likely get lost in selection phase. Conventional GP has no means to counteract that loss. However, traces and execution records introduced in the previous section may reveal such intermediate *behavioral patterns*. Given that, it seems tempting to look for them in order to identify the subprograms that 'relate' to the task in question. Programs that feature such subprograms could be then promoted, to allow the search operators to turn them into better-performing candidate solutions. For instance, a fortunate crossover may mate such a promising subprogram with a piece of code that together leads to optimal solution. This acquired knowledge could be alternatively used more explicitly, for instance by archiving the subprograms and then reusing them via search operators.

A skilled human programmer may discover behavioral patterns and exploit them to design a program that meets the specification of a program synthesis task. Humans in general are known to be incredibly good at spotting and thinking in patterns when solving all sorts of problems — for this reason they have been termed *informavores* (Miller, 1983). A sizeable part of AI research is about mimicking such capabilities (Hofstadter, 1979). Moreover, humans can *anticipate* the patterns that are desirable in a given problem and often use domain and commonsense knowledge for that purpose. Consider the task of synthesizing a program that calculates the median of a list of numbers. The background knowledge tells us that a reasonable first stage of solving this task is to sort the list. In terms of execution records, reaching an intermediate execution state that contains the sorted elements of the list is desirable in this task.

To realize these opportunities, an efficient detector of 'interesting' (relevant for a given program synthesis task) behavioral patterns is necessary. One may for instance analyze how execution traces converge between tests, because this to some extent determines program output — if two or more traces arrive at the same execution state, their further courses must be the same, assuming that an execution state captures everything about the execution environment (by including, for instance, instruction pointer). In (Krawiec and Solar-Lezama, 2014), we proposed an approach that quantified program quality with respect to such convergences of traces, using concepts from information theory.

Nevertheless there exists a wider class of behavioral patterns of potentially greater interest, namely the patterns that are detectable by conventional knowledge discovery and machine learning (ML) algorithms. Such patterns are perused by the method described in the following, termed PANGEA (PAtterN Guided Evolutionary Algorithms), originally proposed in (Krawiec and Swan, 2013) and then extended in (Krawiec and O'Reilly, 2014). Technically, behavioral patterns are being revealed there by a ML algorithm trained on execution traces. Information on the resulting classifier is then used to augment the fitness function. By relying on generic ML tools, this process does not rely on domain knowledge (as is common for humans). Rather, it seeks abstract regularities that can be used to predict the correct output of a program. If this approach is able to reveal meaningful dependencies between partial outcomes and the desired output, we may hope to thereby promote programs with the potential to produce good results in future, even if at the moment of evaluation the output they produce is incorrect.

The ML perspective on behavioral program synthesis originates in the observation that an execution trace bears some similarity to an *example* in ML. Assuming the execution record resulting from applying p to all tests is aligned, i.e., the states in particular traces correspond to each other, the columns of the record can be likened to *attributes* in ML. The desired program output y corresponds in this context to the desired response of a classifier (or regressor, depending on the nature of the task). And crucially, a ML induction algorithm (*inducer* for short), given a set of such examples, can be used to produce a classifier that predicts the desired output of the program based on the attributes describing execution traces.

The method proceeds in the following steps, exemplified in Fig. 1.1:

1. An execution record is built by running the program on the tests.
2. The execution record is transformed into a conventional ML dataset D.
3. A ML induction algorithm is applied to D, resulting in a classifier C.
4. Program evaluation is calculated from the properties of C.

The record built in Step 1, as explained in the example in Sect. 1.2 (Fig. 1.1), is subsequently transformed in Step 2, resulting in the training set labeled as 'ML dataset' in the figure. In this case of simple tree-based GP, the attributes correspond one-to-one to the columns of the execution record, so the only essential change is the addition of the program output y as a dependent variable (class label) in the dataset. Transformation of an execution record into a ML dataset could be more sophisticated, for instance if states represent compound rather than elementary data types. More advanced transformation could facilitate discovery of behavioral patterns, for instance when representation biases of a ML classifier prevent it from capturing certain classes of pattern. Yet another motivation is to allow discovery of higher-order patterns that are unobservable when each attribute reflects a single execution state. Though these options deserve future research, here we focus on tree-based GP and the straightforward, one-to-one transformation of the columns of execution record into ML attributes.

Given the training set D, in Step 3 we train a ML classifier C on it. In (Krawiec and Swan, 2013) and (Krawiec and O'Reilly, 2014), we used the decision tree inducers (C4.5 (Quinlan, 1992) and REP-tree, respectively). For the example in Fig. 1.1, a decision tree induction algorithm produced the classifier labeled 'Decision tree', considering attributes x_i as well as the decision class y as nominal variables. The tree comprises five nodes, uses attributes x_4 and x_1 to predict the output of the program, and commits no errors on D.

1.3.1 Search drivers

The classifier maps the attributes derived from intermediate execution states onto the desired output of the program. In a sense, it attempts to complement the program's capability for solving the problem (i.e. producing the desired output value). This observation motivates the design of specific evaluation functions. If the traces reveal regularities that are relevant for predicting the desired output, then the induction algorithm should be able to build a classifier that is (i) compact and (ii) commits relatively few classification errors. These aspects are strongly related to each other, which we illustrate in the following.

Consider first the case of an optimal program p. p solves the task, i.e. produces the desired output $y_i = p(\mathbf{x}_i)$ for all tests $(\mathbf{x}_i, y_i) \in T$. Since each trace ends with a final execution state, and the attributes are collected from all states, then the last attribute in D will be among them. Because p solves the task, that attribute will be identical to the desired output. In such a case, the induction algorithm may produce a classifier of C that involves only that attribute, e.g. a decision tree composed of a single decision node and k leaves corresponding to the k unique desired outputs. Such a decision tree is thus quite compact and commits no classification errors.

Now consider a non-optimal program. Assume its output diverges so much from the desired output that the corresponding attribute is useless for prediction. In such a case, it is likely for the induced classifier to rely on the other attributes, derived from the intermediate execution states. Individually, such attributes have usually limited predictive power, unless the corresponding column in an execution record happens to capture some key aspect of the task. In consequence, the resulting classifier of C needs to rely on many such attributes and thus may be quite complex. In the case of decision trees, the tree will feature many decision nodes. In general, the size and predictive accuracy of the classifier depend on the degree to which the intermediate states *relate* to the desired output.

These examples illustrate that complexity and predictive capability of a classifier are related to each other in a nontrivial manner. Aggregating them would involve unnecessary loss of information, as we argued earlier. This motivated us to define *two* evaluation functions: the *classification error* and *classifier complexity.* The technical definition of the latter depends on classifier representation; for decision trees, it will be the number of tree nodes. Clearly, neither of these evaluation functions alone captures fully the relatedness of attributes to the desired output. It becomes natural to use them side-by-side. In (Krawiec and Swan, 2013), we aggregated them into a single evaluation function. In (Krawiec and O'Reilly, 2014), we kept them separate and relied on multiobjective approach, employing the Non-dominated Selection Genetic Algorithm (NSGA-II, (Deb et al, 2002)). NSGA-II relies on tournament selection on Pareto ranks to make the choices. To break the ties on ranks, it employs *sparsity*, a measure that rewards the candidate solutions that feature less common scores on criteria. The method is also elitist in selecting from the combined set of parents and offspring (rather than from parents alone).

We postulate that quantities like classifier error and classifier complexity (as well as the information-theoretical measures we proposed in (Krawiec and Solar-Lezama, 2014)) share certain features in common and exemplify a new class of evaluation functions, which we refer to as *search drivers.* A search driver can be considered as a 'quasi' evaluation function. It is expected to provide a certain search gradient towards the global optima, but not necessarily a strong one — we posit that what matters is the *direction* of that gradient rather than its magnitude. We are particularly interested in search drivers that are uncorrelated with the original objective function, as this opens the possibility of using them (or multiple search drivers) together, preferably in a multiobjective setup. Also, we do not expect search drivers to be minimized at the optima — we find this requirement unnecessarily constraining when designing search drivers, while in GP program correctness can be easily verified in abstraction from evaluation function.

In EC, the concept that arguably most resembles that of search driver is *surrogate fitness.* Also known as *approximate fitness function* or *response surface* (Jin et al, 2002), a surrogate function provides a computationally cheaper approximation of the original objective function that comes with a given problem. Search drivers diverge however from surrogate fitness in several respects. Firstly, surrogate functions are by definition meant to *approxi-*

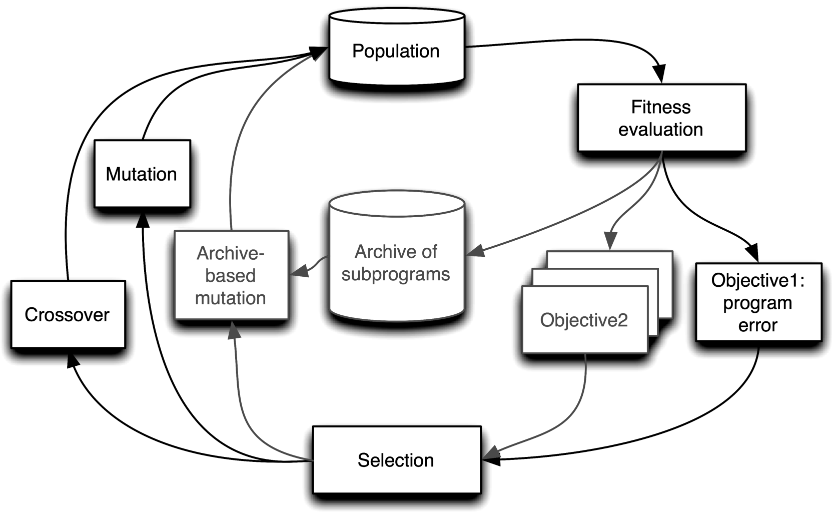


Fig. 1.2 As a side-effect of behavioral evaluation, evaluation can identify useful subprograms in programs being evaluated. Such subprograms can be gathered in an archive, maintained throughout the entire evolutionary run, and reused by search operators (here: archive-based mutation). Empirical evidence shows that such *code reuse* can substantially improve search performance (Krawiec and O'Reilly, 2014).

mate the original objective function. Search drivers lack this intent. Given the challenges plaguing conventional objective functions (see Introduction), why would one want to approximate them? Secondly, search drivers are intended to aid GP meant as a *search*, not optimization problem. This leaves more freedom in their design, which do not have to 'mimic' the objective function across the entire search space. Thirdly, in a program synthesis task, a search driver is not required to be consistent with the objective function in attaining minimal values at global optima. In surrogate fitness, such consistency is essential. And last but not least, a primary rationale for surrogate fitness is high computational cost of original objective function, while the role of search drivers is to help navigate more effectively in the search space.

These differences justify the conceptual distinctness of search drivers. In an ongoing work, we hope to provide a more sound formalization of this concept and come up with guidelines for principled design of search drivers.

1.3.2 Experimental evidence

In (Krawiec and Swan, 2013) and (Krawiec and O'Reilly, 2014) we applied PANGEA to PushGP (Spector and Robinson, 2002) and tree-based

GP respectively. In both cases, the behavioral approach systematically outperformed the configurations driven by conventional fitness functions and control configurations devised to test more specific hypotheses (e.g., which of the abovementioned search drivers is more essential for performance). In the case of tree-based GP, we also extended the approach with *code reuse*: the subprograms indicated as potentially valuable in the process (i.e., corresponding to the attributes used by a decision tree) were retrieved from the evaluated programs, stored in a carefully maintained archive, and reused by an appropriately designed mutation operator. Code reuse lead to further dramatic boosts of performance, measured in terms of success rate, error rate, predictive accuracy, and, interestingly, program size. For instance, on the suite of 35 benchmarks used in (Krawiec and O'Reilly, 2014), the average rank on success rate was 2.43 for PANGEA with code reuse, compared to 3.10 for conventional GP working with 10 times larger population, and 3.86 for GP working with same-sized population (100). Two other PANGEA-based setups, one of them using only two objectives and the other one without archive, ranked 3rd and 4th with average ranks of 3.36 and 3.43, respectively. Two-objective GP working with program error and program size as objectives came last, with the average rank of 4.83. Other performance indicators, like program error and predictive accuracy, were also in favor of behavioral approach. For detailed account on experimental results, see (Krawiec and Swan, 2013) and (Krawiec and O'Reilly, 2014).

1.4 Consequences of behavioral perspective

Complete characterization of program behavior can be a natural means for assessing and controlling the *diversity* of programs. For instance, a selection operator can be easily designed that, given two programs that pass the same number of tests but vary in execution record, allows them co-exist in an evolving population (by, e.g., selecting them both). No dedicated mechanism for controlling or inducing diversity may be necessary – behavioral evaluation *implicitly* provides for phenotypic diversity. This property may help mitigate the risk of premature convergence and overfocusing on local optima. The positive experimental evidence on the performance of behavioral approaches (Section 1.3.2) can be in part attributed to this characteristic. The importance of behavioral diversity has been also corroborated by methods like implicit fitness sharing (Smith et al, 1993; McKay, 2000), co-solvability (Krawiec and Lichocki, 2010), or more recently lexicase selection (Helmuth et al, 2014), where the last one seems to be particularly effective at trading-off diversity maintenance and selective pressure on an evolving population (Liskowski et al, 2015).

Behavioral characterization of programs may also facilitate *task decomposition.* Automatic detection of a task's internal modularity and performing appropriate decomposition has been for long considered one of the main challenges in designing intelligent systems, and is an important area of research in computational and artificial intelligence (Watson, 2006). In behavioral program synthesis, there are at least two alternative avenues to decomposition, both of which can be conveniently explained by means of the execution record.

Firstly, by providing a separate account of program execution *for every test*, execution records open the door to 'horizontal', 'test-wise' task decomposition. This capability is essential also for semantic GP (and indeed other traditional approaches), where some crossover operators combine the behaviors of parents *on particular tests*. This is most evident for exact geometric semantic crossover (Moraglio et al, 2012), especially for the Boolean domain. That operator, when applied to parent programs p_1, p_2, generates a random Boolean subprogram p_r and produces an offspring that combines p_1, p_2 with p_r in a straightforward expression $(p_1 \wedge p_r) \vee (p_2 \wedge \overline{p_r})$. In the offspring, p_r works as a mask: it 'mixes' parents' behaviors by deciding, for each test individually, which parent to copy the output from. For the tests for which p_r returns $true$, the offspring behaves like p_1, and if p_r returns $false$, it behaves like p_2.

The presence of complete execution traces in an execution record facilitates also the less obvious 'vertical' task decomposition. What we mean here is the stage-wise structure of a task, as explained on the example of calculating the median in Section 1.3. In that example, the desired decomposition consists of splitting the original programming task into two separate subtasks of (i) sorting the list and (ii) retrieving the central element of the sorted list. Arguably, solving each of these subtasks separately can be expected to be easier than synthesizing a complete program that calculates the median. We posit that such desired decompositions can be, at least for some programming tasks, automatically derived from a working population of programs by analyzing execution records. In (Krawiec, 2012), we provided some experimental evidence for this hypothesis: 'behavioral trajectories' tend to cluster, thereby revealing the internal structure of a task.

In this chapter, we considered methods that use behavioral information primarily to *drive* the selection process: an alternative evaluation function characterizes (possibly in a multi-objective fashion) the candidate solutions, and that information is used to select the most promising of them. The above remarks on task decomposition point to the alternative ways of exploiting behavioral information, in particular by redefining search operators. The code reuse mutation operator described in Section 1.3.2 and in (Krawiec and O'Reilly, 2014) is an example of such functionality. However, that operator implants the valuable code fragments in the offspring at random locations. Given execution records of mutated/recombined programs, search operators can be even more sophisticated in behavioral terms. For instance, a behaviorally-aware crossover operator could recombine the parents so as to achieve the desired behavioral effect (e.g. by combining a list-sorting subprogram with a subprogram that retrieves the central element from a list in the median problem mentioned earlier).

The behavioral perspective adopted in this chapter in the context of GP has interesting implications beyond program synthesis. One can draw immediate parallels between the trace of stepwise execution of a GP program on a fitness case and the search trajectory of a metaheuristic solver acting on a problem instance. The 'state' of a metaheuristic could of course also include other variables of relevance. For example, the state of Simulated Annealing would include current temperature. In the PANGEA approach described above, a search driver is induced (via a decision tree) from the executable structure. The essential difference in the extension to metaheuristics is that with the GP approach, the executable structure *is* the candidate solution, whereas in this extended approach it is the *means* by which solutions are found (i.e. the particular way in which temperature is modified throughout a Simulated Annealing run). It may nonetheless be possible to obtain search drivers in this more general context by correlating solver state against the candidate solutions representing its current search progress, using any of the gamut of ML techniques mentioned above.

There is much emphasis in the optimization research community on providing solutions for individual problem instances which are 'good enough, quickly enough'. It must not be forgotten that the most significant improvements have arisen from analytical and scientific activity, rather than the engineering activity of 'manual tweaking' of operators and parameters. It is therefore vital to build tools to help distinguish 'universal' features of solvers from 'parochial' ones. The primary strength of GP above other regression techniques is as a model-agnostic mechanism for knowledge discovery. A wider research agenda towards 'robot scientists' (Sparkes et al, 2010) that actively seek correlates between their effectors (e.g. generated metaheuristic search operators) and their observed effects allows these strengths to be directed back into the optimization process itself. This wider agenda of an autonomous search agent capable of metacognitive activity invites contribution from areas such as developmental robotics (Lungarella et al, 2003) and pattern theory (Grenander, 1989). This is of course a different class of activity from optimizing an individual problem instance, but architectures of this general nature (e.g. (Swan et al, 2014)) are necessary in order to automate that which currently requires the labour of skilled researchers.

1.5 Conclusions

The behavioral perspective on program synthesis urges us to rethink the structure and workflow of typical GP algorithm and generic evolutionary methods. A typical iterative optimization algorithm can be visualized as a loop of evaluation phase, selection phase, and the phase of applying search operators ('variation' in evolutionary terms). An evaluation function is typically externalized as a separate component, and communicates only with selected stages of the loop. For behavioral approach 'in the large', it may be more appropriate to visualize the workflow as a *network of interconnected components* that exchange information about the search process. By having access to behavioral characteristics of candidate solutions, the components in such an architecture would be more empowered when making decisions about the fate of particular candidate solutions.

To an extent, the behavioral approach can be seen as a means for making search process more 'intelligent' while keeping it relatively ignorant about the domain-specific aspects. By observing program behavior as captured in an execution record, a search algorithm gains better insight into program specifics, while abstracting from characteristics of the underlying program representation, programming language, etc. For instance, PANGEA may observe similar or even the same execution records whether the evolving programs implement imperative or functional programming paradigms.

The fascinating realization is that there are probably many potentially useful search drivers beyond the conventional ones, and beyond the ones discussed in this chapter. It is even possible to that some of them may provide better performance of search algorithms than anything known to date. In this chapter and previous works on this topic, we have only scratched the surface of how search drivers can be defined (or automatically derived from a problem (Kocsis and Swan, 2014)). In a longer-term research perspective, it would be highly desirable to come up with a principled approach to the design of search drivers.

Acknowledgements Krzysztof Krawiec acknowledges support from grant 09/91/DSPB/0572 and National Science Centre grant 2014/15/B/ST6/05205. Una-May O'Reilly acknowledges support from Li Ka Shing Foundation.

References

Brameier M, Banzhaf W (2007) Linear Genetic Programming. No. XVI in Genetic and Evolutionary Computation, Springer, URL http://www.springer.com/west/home/default?SGWID=4-40356-22-173660820-0

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. Evolutionary Computation, IEEE Transactions on 6(2):182 –197, DOI 10.1109/4235.996017

Grenander U (1989) Advances in pattern theory. Ann Statist 17(1):1–30, DOI 10.1214/aos/1176347002, URL http://dx.doi.org/10.1214/aos/1176347002

Helmuth T, Spector L, Matheson J (2014) Solving uncompromising problems with lexicase selection. Evolutionary Computation, IEEE Transactions on

PP(99):1–1, DOI 10.1109/TEVC.2014.2362729

Hofstadter DR (1979) Godel, Escher, Bach: An Eternal Golden Braid. Basic Books, Inc., New York, NY, USA

Jin Y, Olhofer M, Sendhoff B (2002) A framework for evolutionary optimization with approximate fitness functions. IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION 6:481–494

Kocsis ZA, Swan J (2014) Asymptotic Genetic Improvement Programming with Type Functors and Catamorphisms (extended abstract). In: Johnson C, Krawiec K, Alberto Moraglio aMO (eds) Semantic Methods in Genetic Programming (SMGP) at Parallel Problem Solving from Nature (PPSN XIV), Ljubljana, Slovenia

Krawiec K (2012) On relationships between semantic diversity, complexity and modularity of programming tasks. In: Soule T, Auger A, Moore J, Pelta D, Solnon C, Preuss M, Dorin A, Ong YS, Blum C, Silva DL, Neumann F, Yu T, Ekart A, Browne W, Kovacs T, Wong ML, Pizzuti C, Rowe J, Friedrich T, Squillero G, Bredeche N, Smith S, Motsinger-Reif A, Lozano J, Pelikan M, Meyer-Nienberg S, Igel C, Hornby G, Doursat R, Gustafson S, Olague G, Yoo S, Clark J, Ochoa G, Pappa G, Lobo F, Tauritz D, Branke J, Deb K (eds) GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference, ACM, Philadelphia, Pennsylvania, USA, pp 783–790, DOI doi:10.1145/2330163.2330272

Krawiec K, Lichocki P (2010) Using co-solvability to model and exploit synergetic effects in evolution. In: Schaefer R, Cotta C, Kolodziej J, Rudolph G (eds) PPSN 2010 11th International Conference on Parallel Problem Solving From Nature, Springer, Krakow, Poland, Lecture Notes in Computer Science, vol 6239, pp 492–501, DOI doi:10.1007/978-3-642-15871-1_50

Krawiec K, O'Reilly UM (2014) Behavioral programming: a broader and more detailed take on semantic GP. In: Igel C, Arnold DV, Gagne C, Popovici E, Auger A, Bacardit J, Brockhoff D, Cagnoni S, Deb K, Doerr B, Foster J, Glasmachers T, Hart E, Heywood MI, Iba H, Jacob C, Jansen T, Jin Y, Kessentini M, Knowles JD, Langdon WB, Larranaga P, Luke S, Luque G, McCall JAW, Montes de Oca MA, Motsinger-Reif A, Ong YS, Palmer M, Parsopoulos KE, Raidl G, Risi S, Ruhe G, Schaul T, Schmickl T, Sendhoff B, Stanley KO, Stuetzle T, Thierens D, Togelius J, Witt C, Zarges C (eds) GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation, ACM, Vancouver, BC, Canada, pp 935–942, DOI doi:10.1145/2576768.2598288, URL http://doi.acm.org/10.1145/2576768.2598288, best paper

Krawiec K, Solar-Lezama A (2014) Improving genetic programming with behavioral consistency measure. In: Bartz-Beielstein T, Branke J, Filipic B, Smith J (eds) 13th International Conference on Parallel Problem Solving from Nature, Springer, Ljubljana, Slovenia, Lecture Notes in Computer Science, vol 8672, pp 434–443, DOI doi:10.1007/978-3-319-10762-2_43

Krawiec K, Swan J (2013) Pattern-guided genetic programming. In: Blum C, Alba E, Auger A, Bacardit J, Bongard J, Branke J, Bredeche N, Brockhoff D, Chicano F, Dorin A, Doursat R, Ekart A, Friedrich T, Giacobini M, Harman M, Iba H, Igel C, Jansen T, Kovacs T, Kowaliw T, Lopez-Ibanez M, Lozano JA, Luque G, McCall J, Moraglio A, Motsinger-Reif A, Neumann F, Ochoa G, Olague G, Ong YS, Palmer ME, Pappa GL, Parsopoulos KE, Schmickl T, Smith SL, Solnon C, Stuetzle T, Talbi EG, Tauritz D, Vanneschi L (eds) GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference, ACM, Amsterdam, The Netherlands, pp 949–956, DOI doi:10.1145/2463372.2463496

Liskowski P, Krawiec K, Helmuth T, Spector L (2015) Comparison of semantic-aware selection methods in genetic programming. In: Proceedings of the seventeenth annual conference on Genetic and Evolutionary Computation Companion, GECCO Comp, (accepted)

Lungarella M, Y GM, Z RP, S G, Y I (2003) Developmental robotics: a survey. Connection Science 15:151–190

McKay RIB (2000) Fitness sharing in genetic programming. In: Whitley D, Goldberg D, Cantu-Paz E, Spector L, Parmee I, Beyer HG (eds) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), Morgan Kaufmann, Las Vegas, Nevada, USA, pp 435–442, URL http://www.cs.bham.ac.uk/ wbl/biblio/gecco2000/GP256.pdf

Miller GA (1983) Informavores, Wiley-Interscience, p 111–113

Moraglio A, Krawiec K, Johnson CG (2012) Geometric semantic genetic programming. In: Coello Coello CA, Cutello V, Deb K, Forrest S, Nicosia G, Pavone M (eds) Parallel Problem Solving from Nature, PPSN XII (part 1), Springer, Taormina, Italy, Lecture Notes in Computer Science, vol 7491, pp 21–31, DOI doi:10.1007/978-3-642-32937-1_3

Popovici E, Bucci A, Wiegand RP, de Jong ED (2011) Handbook of Natural Computing, Springer-Verlag, chap Coevolutionary Principles

Quinlan J (1992) C4.5: Programs for machine learning. Morgan Kaufmann, San Mateo

Smith R, Forrest S, Perelson A (1993) Searching for diverse, cooperative populations with genetic algorithms. Evolutionary Computation 1(2)

Sparkes A, Aubrey W, Byrne E, Clare A, Khan M, Liakata M, Markham M, Rowland J, Soldatova L, Whelan K, Young M, King R (2010) Towards robot scientists for autonomous scientific discovery. Automated Experimentation 2(1):1, DOI 10.1186/1759-4499-2-1, URL http://www.aejournal.net/content/2/1/1

Spector L, Robinson A (2002) Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines 3(1):7–40, DOI doi:10.1023/A:1014538503543, URL http://hampshire.edu/lspector/pubs/push-gpem-final.pdf

Swan J, Woodward J, Özcan E, Kendall G, Burke E (2014) Searching the hyper-heuristic design space. Cognitive Computation 6(1):66–73, DOI 10.1007/s12559-013-9201-8

Tomassini M, Vanneschi L, Collard P, Clergue M (2005) A study of fitness distance correlation as a difficulty measure in genetic programming. Evolutionary Computation 13(2):213–239, DOI doi:10.1162/1063656054088549,

Watson RA (2006) Compositional Evolution: The impact of Sex, Symbiosis and Modularity on the Gradualist Framework of Evolution, Vienna series in theoretical biology, vol NA. MIT Press, URL http://eprints.ecs.soton.ac.uk/10415/