2

Limitations of conventional program evaluation

In Sect. 1.4, we identified several challenges for program synthesis, among others the vastness of search space and the intricate way in which program code determines the effects of computation. In this chapter, we identify and discuss the consequences of the conventional approach to program evaluation in generative program synthesis. Though focused mostly on the generative paradigm of GP, some of the observations made here apply to other generative synthesis approaches.

2.1 Evaluation bottleneck

As introduced in Sect. 1.5.3, the evolutionary search process in conventional GP is driven by an evaluation function (fitness) $f(p)$ applied to candidate programs $p \in P$. Evaluation typically boils down to counting the number of tests failed or passed by p. For convenience we repeat here the formula for the conventional evaluation function, i.e. the objective function (1.7):

$$f_o(p) = |\{(in, out) \in T : p(in) \neq out\}|. \tag{2.1}$$

Assessing candidate solutions using a scalar performance measure has several merits. First of all, it is in a sense minimal – it is hard to imagine a simpler way of informing a search algorithm about the solutions' characteristics. It is also compatible with the conventional way of posing problems in optimization and machine learning. Last but not least, it eases separation of generic search algorithms from domain-specific evaluation functions, which is so essential for *meta*heuristics. No wonder that this 'design pattern' is so common that we rarely ponder over its other consequences.

Unfortunately, there is a price to pay for all these conveniences, a price that stems from the inevitable loss of information that accompanies scalar evaluation. Programs are complex combinatorial entities and program execution is

K. Krawiec, *Behavioral Program Synthesis with Genetic Programming*,
Studies in Computational Intelligence 618,
DOI: 10.1007/978-3-319-27565-9_2

a nontrivial process. Yet, in conventional GP all what is left of that process is a single number, i.e. the number of failed tests for discrete domains (1.7) or a continuous error in the case of regression problems (e.g., (1.8)).

main claim of this book

The main tenet of this book is that the conventional scalar evaluation *denies a search algorithm access to detailed behavioral characteristics of a program*, while *that information could help to drive the search process more efficiently*. This observation can be alternatively phrased using the message-passing metaphor typical for 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. And compressing all information about program behavior into a single number is inevitably lossy.

To illustrate the chasm between the richness of program behaviors and the paucity of scalar evaluations, consider again synthesis of Boolean programs. Assume for the sake of argument that we identify program behavior with the combination of outputs it produces for all tests (as in semantic GP, Chap. 5). There are 2^{2^k} such behaviors of k-ary Boolean programs. For the 11-bit multiplexer example considered in Sect. 1.4), this is 2^{2048}. On the other hand, the objective function for this problem assumes only $2^{11} + 1 = 2049$ distinct values (0 to 2048 inclusive), i.e. can be represented with 11 bits (excluding the correct program). To believe that a search algorithm can efficiently search a space of 2^{2048} behaviors by obtaining 11 bits of information for each visited candidate solution is very optimistic, if not naive – and recall that we consider the Boolean domain, arguably the simplest one. It is hard to resist quoting the classic reflection on the size of the Universe:

> Gigantic multiplied by colossal multiplied by staggeringly huge is the sort of concept we're trying to get across here. [1]

evaluation bottleneck

The existence of this *evaluation bottleneck* is confirmed by practice. For instance, even though contemporary GP algorithms manage to synthesize 11-bit multiplexers, the parity problem with the same number of variables is for them very difficult and hardly ever gets solved.

2.2 Consequences of evaluation bottleneck

In this section, we discuss the implications of scalar evaluation that originate in the aggregation of outcomes of multiple interactions of a program with the tests. Such aggregative functions prevail in generative program

synthesis, but have been also intensely studied in *test-based problems* [16, 24] (Sect. 4.1). A function that counts the failed tests (2.1) is the most common, additive representative of that class. The error functions commonly used in symbolic regression (like MSE (1.8) or Mean Absolute Deviation, MAD) also belong to this class, as does the less common multiplicative aggregations.

2.2.1 Discreteness and loss of gradient

Many evaluation functions used in GP are by nature *discrete*. The objective function f_o that counts the failed tests is quite an obvious example here (2.1): f_o can assume only $|T| + 1$ unique values. When candidate programs pass the same number of tests, f_o fails to provide a search gradient. This is particularly likely when T is small, which is often practiced in GP to reduce computational cost. However, having many tests does not necessarily solve this problem either, because programs in the population improve and with time it becomes more likely for them to pass the same number of tests. Also, in some domains and for some problems, achieving certain values of f_o is more likely than others; for instance the parity problem (see Chap. 10 for definition) is notorious for many programs having f_o that is a power of two. In such situations, the evaluation function (and consequently a selection operator) ceases to differentiate candidate solutions, the search gradient is lost, and search becomes purely random.

loss of gradient

Increasing the number of tests is not necessarily beneficial for yet another reason. Consider a set T of $\mathcal{T} = 10\,000$ tests and two program synthesis tasks defined for that problem, with evaluation functions based on randomly drawn subsets $T_1, T_2 \subset \mathcal{T}$ of, respectively, 100 and 1 000 tests. T_2 is ten times larger, so the values of an evaluation function that drives the corresponding search process are more *precise*, i.e. estimates better the true performance. But in the presence of a rugged fitness landscape (Sect. 1.4), better precision may be insignificant, and an evaluation function based on T_1 may perform equally well.

The problem of discreteness in principle does not apply to evaluation functions that aggregate continuous interaction outcomes (e.g., (1.8)), typically used in regression problems. However, due to the many-to-one mapping from programs to evaluations and limited number of tests, there are usually many programs that implement the same real-valued function. Such candidate solutions will receive identical evaluation and become indistinguishable, or worse, the presence of rounding errors will render one of them slightly better, leading to an unintended search bias.

2.2.2 Compensation

Another consequence of adopting an aggregative objective function like the objective function f_o is *compensation*: all programs that pass the same number of tests are considered equally fit, no matter *which* particular tests they pass. In other words, such a function is *symmetric* with respect to tests[1]. This characteristic ignores the fact that some tests can be *inherently* more difficult than others: in a given programming language $\mathcal{P}$, more programs may exist that pass a test t_1 than programs that pass some some other test t_2. Following [69], we can formalize the difficulty of a test $t = (in, out) \in \mathcal{T}$ for a programming language $\mathcal{P}$ as

$$diff(t) = diff((in, out)) = \Pr(p \in \mathcal{P} : p(in) \neq out). \tag{2.2}$$

objective test difficulty

subjective test difficulty

We further refer to this quantity as to *objective test difficulty*. $diff(t)$ is thus the probability that a program randomly picked from $\mathcal{P}$ passes test t. Alternatively, one might define *subjective test difficulty*, i.e. the probability of passing a test by a program synthesized by a given search algorithm. Nevertheless, objective test difficulty $diff(t)$ is more universal and can be estimated even if $\mathcal{P}$ is large or infinite [69].

It is interesting to realize that encountering in the real world a problem with *diff* varying across tests in $\mathcal{T}$ is actually more likely than all tests being equally difficult. This is evidenced not only in program synthesis, but also in test-based problems (Sect. 4.1), e.g., in games [176].

2.2.3 Biased search

Discreteness and compensation are inherent properties of aggregative evaluation functions. More subtle consequences of scalar evaluation come to light only when considering the entirety of the program synthesis process. In the following, we demonstrate the interplay of characteristics of search operators with an objective function. To that end, we employ the formalism of *outcome vector*, i.e. the vector of the outcomes of program's interactions with particular tests:

outcome vector

$$o_T(p) = ([p(in) = out])_{(in,out)\in T}, \tag{2.3}$$

where $[\,]$ is the Iverson bracket, defined for a logical predicate α as

$$[\alpha] = \begin{cases} 1, & \text{if } \alpha \text{ is true} \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

The ith element of $o_T(p)$ is 1 if p passes the ith test in T, and 0 otherwise. As signaled earlier, T needs to be a list, not a set, for o_T to be well-defined.

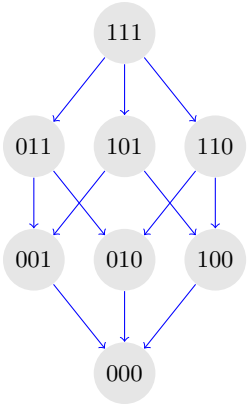

Fig. 2.1: The lattice of outcome vectors for an abstract problem featuring three tests. Value 1 denotes passing a given test, 0 – failing it. Blue arrows mark the dominance relation between outcome vectors.

behavioral descriptor

Notably, an outcome vector is the first non-trivial *behavioral descriptor* the reader encounters in this book.

poset

lattice of outcome vectors

For a given T, the set of all possible outcome vectors forms a special case of partially ordered set (*poset*), a *lattice*. The lattice has $2^{|T|}$ nodes; Figure 2.1 presents an example for three tests. The top element in the lattice corresponds to the target of search (1.4) and groups optimal solutions, i.e. programs that pass all tests and thus achieve $f_o(p) = 0$. The bottom element correspond to the worst solutions ($f_o(p) = |T|$). The outcome vectors in the intermediate layers have evaluation varying from $|T| - 1$ (second layer from the bottom) to 1 (second layer from the top). A generate-and-test program synthesis method like GP will usually start with the programs occupying middle levels in the lattice, and progress toward the top node.

Any program interacting with the tests in T can be unambiguously assigned to one and only one node in this lattice based on its outcome vector. Multiple programs may occupy the same node in the lattice due to the mapping from programs to their behaviors being many-to-one (Sect. 1.4). For most programming languages $\mathcal{P}$, some behaviors of programs in $\mathcal{P}$ are more common than others. As a result, the distribution of programs over lattice nodes is usually highly non-uniform (see [106, Chap. 7] for an example for the Boolean domain).

The arcs in Fig. 2.1 illustrate the *dominance relation* between outcome vectors. An arc connects a node o_1 to o_2 if and only if every test passed in o_1 is also passed in o_2 *and* there is exactly one test in o_2 that is not passed in o_1. However, these arrows have little to do with possible moves of a search

[1] Unless it differentiates their importance in some way (e.g., by weighing)

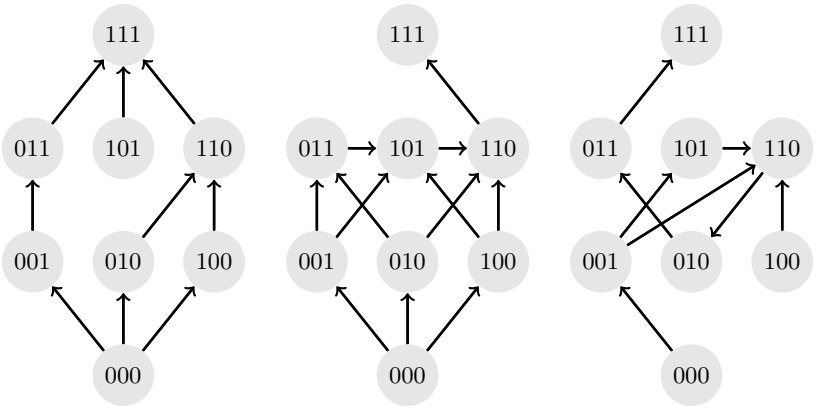

Fig. 2.2: Transition graphs on outcome vectors for three abstract problems, each featuring three tests. Contrary to Fig. 2.1 where arrows mean dominance, here they mark the changes of outcome vectors (behavioral, phenotypic changes) resulting from modifications applied to programs by a search algorithm (genotypic changes).

algorithm. How a given program synthesis algorithm traverses such a lattice is in general unrelated to the dominance relation. The reason for that is the highly non-local genotype-phenotype mapping discussed in Sect. 1.4. A search operator may not be able to improve a current candidate program (with an outcome vector in o_2) by augmenting it with the capability of solving yet another test while preserving that capability for the already passed tests (so that it ends up in o_1). On the other hand, the same search operator may be capable of flipping *multiple* elements of the outcome vector of a given program in a single application, moving a program several levels up, down, or sideways in the lattice.

Therefore, in the figures that follow we drop the dominance edges and represent only the transitions between outcome vectors as induced by a given search operator. Though the spatial arrangement of nodes remains the same, these posets are not lattices anymore and will be referred to as *transition graphs*. Similar graphs have been previously considered in [60], however without taking into account the spatial arrangement of nodes resulting from stratification of evaluation.

transition graph

Example 2.1. Consider a hypothetical task with such a transition graph shown in Fig. 2.2a. The arrows mark the possibility of transitions between outcome vectors. For instance, the arc from the node labeled 010 to the node labeled 110 indicates that there exists a program with the outcome

vector 010 such that it can[2] be modified by the considered search operator so that its outcome vector changes to 110. Note that in general one should not expect the arrows to be mirrored: a search operator may be unable to revert the effects of its application.

It does not take long to realize that the problem shown in Fig. 2.2a is easy: the transitions are aligned along the gradient of the objective function f_o, so even a straightforward search algorithm (e.g., greedy search) should easily traverse the path from 000 to 111. Now consider the problem shown in Fig. 2.2b: here, the transition from 101 to 110 is not accompanied by an improvement (f_o remains to be 1). Because the evaluation function imposes only a vertical gradient in the graph, if a search algorithm that accepts only strict improvements happens to visit 101, it will get stuck there. This does not seem to be much of a problem for stochastic search algorithms like GP, which could still move from 101 to 110 by pure chance. Consider however the problem shown in Fig. 2.2c. Once a search process reaches the combination 110, further progress can be made only by moving to 010, which implies the deterioration of f_o from 1 to 2. Only search algorithms capable of accepting such deterioration will be able to overcome this trap. Again, stochasticity of GP will occasionally permit that, but the likelihood of such an event may vary, and such transition graphs are in general harder to traverse than those shown in Figs. 2.2a and 2.2b. ■

fitness landscape

To an extent, the above graphs are related to *fitness landscapes* [194]. Fitness landscapes visualize a scalar evaluation function stretching over a solution space $\mathcal{P}$ arranged with respect to the neighborhood induced by a search operator. The case in Fig. 2.2a is characteristic for a unimodal fitness landscape devoid of *plateaus*, i.e. neighboring solutions with the same fitness. The case in Fig. 2.2b reveals presence of some plateaus (e.g., (011,101,110)), and the one in Fig. 2.2c features *traps* (110,101) and can be thus characterized as *deceptive*.

behavioral equivalence class

This is however where the analogy with fitness landscapes ends. A point on a fitness landscape corresponds to one candidate solution. The nodes in Fig. 2.2 correspond to behavioral *equivalence classes* induced by outcome vectors. By this token, they provide more detailed behavioral information and can prospectively allow for identifying more nuanced structures and challenges to them. Recall however that a single node gathers here *multiple* programs (some of them possibly syntactically different) that have the same behavior. That behavior is the only thing they have in common: some of those programs may be able to undergo certain modifications of a search operator, while some not. In this sense, these graphs paint an overly

[2] 'Can', because the outcome of an application of a *stochastic* search operator to a given program is non-deterministic.

optimistic picture about the possible traversals in the space of outcome vectors.[3]

2.3 Experimental demonstration

The examples in Fig. 2.2 are intentionally simple and abstract from a specific domain. In practice, the behavior of a given search algorithm on a given problem is more complex. The transitions between outcome vectors, rather than being possible or impossible, become more or less *likely* due to various biases of search operators. In GP, another reason for that is the stochastic nature of search operators. It is thus justified to ask: are the characteristics discussed above and exemplified in Fig. 2.2 purely hypothetical or do they occur in real-world domains? To answer this question, we experimentally construct analogous graphs for selected real-world problems.

We consider the Boolean domain and programs composed of instructions $\{and, or, nand, nor\}$. To construct a graph, we first generate a sample of 10 000 000 random programs of depth up to 8 using the conventional ramped half-and-half method [79]. For each generated program p, we mutate it, obtaining a program p', calculate the outcome vectors $o(p)$ and $o(p')$, and collect the statistics of transitions between outcome vectors over the entire sample. We employ the subtree-replacing mutation operator commonly employed in GP [148], which uses ramped half-and-half with depth limit 8 to generate random subtrees.

Figure 2.3 visualizes the resulting transition graph for the Boolean 3-bit multiplexer problem (MUX3). In this problem, program input comprises three variables: the first one serves as the 'address line' that decides which of the remaining two input variables should be passed to the output – this is the desired behavior of a correct program. Given three tests, the objective function (1.7) varies from 0 to 8 inclusive. Even though MUX3 is trivial for contemporary program synthesis methods, this small domain is already quite rich in terms of program behavior: there are $2^8 = 256$ possible outcome vectors (and thus nodes in the lattice), and the central layer of the lattice features $\binom{8}{4} = 70$ nodes. For this reason, Fig. 2.3 shows only the three top layers of the lattice, i.e. the outcome vectors that correspond to evaluation 0 (optimal solutions, one node in the top layer), 1 ($\binom{8}{1} = 8$ nodes in the second layer) and 2 ($\binom{8}{2} = 28$ nodes in the third layer). The topmost layers are critical for the success of program synthesis, as in practice search often gets stuck with one or two failing tests and therefore does not make

[3] This leads to an interesting observation concerning the nodes without improving outgoing arrows: although many programs that occupy them are syntactically different, they are all hard to improve.

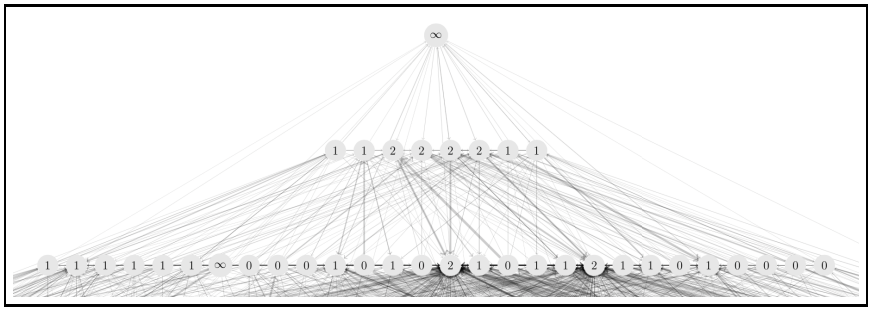

Fig. 2.3: The top three layers of the transition graph on outcome vectors for the MUX3 problem. See [82] for the complete figure.

further progress. The reader is invited to download the image of the entire lattice provided online at [82] to conveniently zoom into details.

The spatial arrangement of the nodes in Fig. 2.3 is analogous to that in Fig. 2.2. This time however the widths of the arrows vary and reflect the estimated probability of transitions. Given a node o, the widths of the outgoing arcs are proportional to the probabilities of mutation moving a program with outcome vector o to successor nodes. For clarity, we do not draw the arcs corresponding to neutral mutations (which would start and end in the same node) and arcs that account for less than 1 percent of outgoing transitions.

Because we are not going to inspect specific outcome vectors, we do not label the nodes with them but with the estimated log-odds of improvement. For a given node, the label is

$$-\left\lfloor \log_{10} \frac{n^-}{n^+} \right\rfloor , \tag{2.5}$$

where n^- is the number of non-improving moves outgoing from that node, and n^+ is the number of improving moves outgoing from the node. For instance, a label '2' indicates that the odds of non-improving moves to the improving ones is of the order of $10^2 : 1$, i.e. $100 : 1$. The ∞ symbol labels the nodes for which no improving move has been found in the sample.

The log-odds clearly increase with improving evaluation f_o, i.e. with the decreasing number of failed tests (1.7). As observed in typical GP runs, with closing to the target (corresponding to the top node of the graph), it becomes harder to make improving moves. This trend becomes even more evident when confronted with the lower layers not shown in print. Nevertheless, the chances of reaching the optimal solution in this example are clearly not negligible, as evidenced by the large number of the arcs incoming to the top node of the lattice. This is due to simplicity of MUX3.

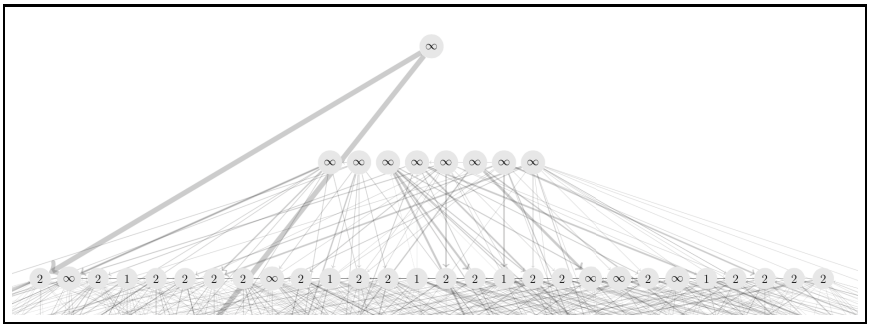

Fig. 2.4: The top three layers of the transition graph on outcome vectors for the PAR3 problem. See [82] for the complete figure.

In Fig. 2.4, we present an analogous graph for the much harder PAR3 problem, where a correct program should return *true* if and only if the number of input variables that have the value *true* is even. The graph, generated using the same instruction set and identical settings as for MUX3, features a substantially different structure of arcs. The log-odds of the orders between 1 and 2 occur already one layer lower than for MUX3. In the second layer from the top, all nodes are marked by ∞ and all the outgoing arrows point downwards: no program with these outcomes vectors in the considered sample underwent an improving mutation.

As a side remark, it is interesting to note that, for binary output domains ($|\mathcal{O}| = 2$), there is one-to-one correspondence between the outputs of a program ($true, false$) and test outcomes $(0, 1)$. Therefore, other things being equal, Figs. 2.3 and 2.4 present, as a matter of fact *the same* transition graph, however with node locations permuted spatially, so that layers group the nodes that have the same evaluation in a given problem. More generally, this graph is common for *all* 3-bit Boolean problems.

2.4 Discussion

The conclusion following from Figs. 2.3 and 2.4 is that *programs' odds for being improved vary by orders of magnitude*. Moreover, the outcome vectors that offer the highest chance of further improvement are not necessarily the easiest ones to attain. For instance, the nodes labeled with 1 in the second layer of Fig. 2.3 ($10:1$ odds of non-improving transitions) do not seem to feature more incoming arcs than the less attractive nodes, i.e. those labeled with 2 (odds $100:1$).

Scalar evaluation performed by the objective function does not reveal such differences. Programs with outcome vectors in the same graph layer receive

the same evaluation and render themselves indistinguishable. Conventional evaluation functions combined with biases of search operators favor certain paths of accretions of *skills*, meant here as capability to pass particular tests. As a result of that bias, certain outcome vectors become particularly easy to attain. If a given outcome vector does not offer an opportunity for further improvement, it forms a counterpart to the conventional notion of *local optimum*. In a longer run, the programs with easy-to-attain outcome vectors tend to prevail in a population, causing it to converge prematurely.

skill

local optimum

premature convergence

Stochastic approaches to program synthesis like GP are *in principle* resistant to premature convergence, because a well-designed stochastic search algorithm visits in the limit *all* points in the reachable search space. However, guarantees 'in the limit' are of little use in practice. In general, GP does not scale well with the growing number of tests and complexity of a synthesis task. It seems thus justified to address the above issues by *making the search algorithm more aware about the differences in program behavior*. This is the revolving theme of this book, and the following chapters review and propose practical means to this end.

Another upshot is that it is not the *number* of tests passed, but the particular *combination* of them that may be critical for a search algorithm to solve a program synthesis task. The interplay of scalar evaluation function with the characteristics of search operators may lead to a situation in which a candidate solutions may need to first master some skills before attempting to master others. This intuition gave rise to some of the GP extensions presented in the next chapters. It was also present in the studies on test-based problems and coevolutionary algorithms [16, 24, 149].

2.5 Related concepts

The limited capability of scalar evaluation functions has been identified in many branches of computational intelligence and beyond. Within evolutionary computation, this problem is usually tackled with techniques. Implicit fitness sharing (Sect. 4.2) and novelty search (Sect. 9.10) are among them. Other approaches include niching [118] and island models [191].

diversity maintenance

The realization of the limited 'informativeness' of the evaluation function has also surfaced beyond evolutionary computation. Deep learning, responsible for the recent progress in the field of artificial neural networks is, at least to some extent, motivated by this observation. Let us quote here one of the most often cited papers in that domain:

deep learning

> Training a deep network to directly optimize only the supervised objective of interest (for example the log probability of correct classification) by gradient descent, starting from random initialized

> parameters, does not work very well. (. . .) What appears to be key is using an additional unsupervised criterion to guide the learning at each layer. [185]

The 'additional unsupervised criterion' is intended to provide an extra (or alternative) guidance for a learning algorithm (an analog to search algorithm in GP). In the cited work, it helps to learn higher-level representations of input data while preserving their key features. Interestingly, this is analogous to the approach we proposed in [100] and present in Chap. 6, where evolving programs are promoted for preserving the relevant information contained in tests at intermediate stages of program execution (which may be likened to hidden layers in a multilayer neural network). Crucially, in doing so we do not actually specify *what* a program should produce at an intermediate execution state. In this sense, we guide thus a search process in an unsupervised way, as in the above quote.

Last but not least, it may be illuminating here to adopt for a moment the biological perspective, the source of inspiration for EC and GP. The possibility of an evaluation function failing to efficiently drive a search process should not come as a surprise, because fitness in biology is not meant to *drive* anything: it is a figment of the conceptual framework in which we phrase the evolution in Nature. Fitness reflects the reproductive success of a given individual (see, e.g., Sect. 2.2 in [139]), and as such is known only a posteriori, once the parents have been selected (for *absolute* fitness) or once the statistics on the descendants of the current individuals are known (for *relative fitness*). And although it summarizes the entirety of individual's skills that are relevant for gaining an evolutionary advantage, it cannot be reverse-engineered: it is virtually impossible to guess from a fitness value *which* skills made a given individual successful.

This observation is related to the fact that the natural evolution does not optimize for anything; rather than that, it opportunistically improves the skills (and combinations thereof) that increase the odds of survival (or comes up with completely new skills). For this reason, the EC paradigm that should be considered the closest to the biological archetype is[4] that of coevolutionary algorithms, where individuals (or species, or individuals and their environments) impose selective pressure on each other, rather than such a pressure originating in some external objective function (Sect. 4.1).

Another biological aspect worth mentioning here concerns the relationship between the aggregative nature of conventional evaluation functions used in EC and *gradualism*, the cornerstone of classical Darwinism (see also the comments on *accretion* in Sect. 1.5.3). As gradualism posits that progress in natural evolution is slow and steady, so passing each new test gives an

[4] Apart from the open-ended evolution, which is however less common for EC and more for artificial life.

individual in EC only a minute evolutionary advantage (unless the number of tests is low, which is of no interest here). However, even if indeed the biological evolution was gradual in this sense (though it has been questioned many times), we posit that enforcing an analogous approach in EC is not necessarily effective. Humans rarely solve problems gradually, case by case. Instead, they identify regularities and perform conceptual leaps via inductive or deductive reasoning. Interestingly, the temporal dynamics of this process can be likened to *punctuated equilibria*, i.e. long periods of no improvement interlaced with sudden rises in fitness, observed both in natural evolution and EC [186].

punctuated equilibria

2.6 Summary and the main postulate

This chapter gathered the evidence that *evaluation bottleneck* resulting from reliance on conventional scalar evaluation functions has detrimental consequences, including loss of search gradient and premature convergence. Arguably, there are domains where an evaluation function is by definition 'opaque' and makes this bottleneck inevitable. For instance, in Black Box Optimization, the value of the evaluation function is the only information on candidate solutions available to a search algorithm. Similarly, *hyper-heuristics* [18] usually observe the *domain barrier* that parts the search algorithm from a problem.

hyper-heuristics

domain barrier

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. This is particularly true for program synthesis, where an act of evaluating a candidate program produces detailed information that can be potentially exploited. There are at least two levels of detail involved in there. Firstly, a program interacts with *multiple* tests. Secondly, a program's confrontation with a single test involves executing *multiple instructions*, each of them having possibly nontrivial effects.

The main claim of this book is that *the habit of driving search using an objective function alone (especially a scalar one) cripples the performance of search algorithms* in many domains (in particular in generative program synthesis), and it *should be abandoned in favor of more sophisticated and more informative* search drivers. By providing search algorithms with more detailed information on program behavior we hope to broaden the evaluation bottleneck and improve their performance.

main claim

It may be worth mentioning that evaluation bottleneck is also awkward in architectural terms, i.e. when looking at a program synthesis system as a network of interconnected components, shown for GP in Fig. 1.1. Why compress evaluation outcomes in one component (evaluation function)

and then force another component (e.g., selection operator) to 'reverse-engineer' them? There are no other reasons for this other than the convention inherited from metaheuristic optimization and excessive adherence to – not necessarily accurate, as we argued in Sect. 2.5 – evolutionary metaphor.

In the following chapters, we present two mutually non-exclusive avenues toward that goal. The first, represented in this book by semantic genetic programming (Chap. 5), consists of designing search operators that produce offspring in a more 'directed' way. The other approach is exercised here more extensively and aims at alternative multifaceted characterizations of program behavior. This philosophy is embodied by implicit fitness sharing and related methods (Chap. 4), which analyze convergence of execution traces (Chap. 6), and pattern-guided program synthesis (Chap. 7). The next chapter introduces the common formal framework that unifies those approaches and facilitates their presentation.

http://www.springer.com/978-3-319-27563-5

Behavioral Program Synthesis with Genetic Programming

Krawiec, K.

2016, XXI, 172 p. 25 illus., 15 illus. in color., Hardcover
ISBN: 978-3-319-27563-5