9

Search drivers

In this chapter, we provide a unified perspective on the methods presented in Chaps. 4–8, the key consequence of which is the concept of *search driver* detailed in Sect. 9.3.

9.1 Rationale for the unified perspective

In Chaps. 4–7, we presented several evaluation methods for characterizing candidate programs in GP. We summarize them in Table 9.1 and contrast with the conventional GP for reference.[1]

The approaches are founded on various formalisms. They rely on different parts of the execution record. Most of them evaluate programs in absolute terms, but some are relative and contextual, i.e. their assessments depend also on the other candidate solutions in a population. In the listed order, they are conceptually more and more sophisticated, and tend to elicit more information from an execution record. The last two listed methods can serve as subprogram providers (Sect. 8.1), i.e. can identify potentially valuable code pieces in evaluated programs.

These differences notwithstanding, all these alternatives to standard GP have been designed with a more or less explicit intention of broadening the evaluation bottleneck and acquiring alternative (or additional) *behavioral information* from candidate solutions (programs). We postulate that by this token they deserve a common conceptual umbrella, and from now on we refer to the evaluation functions they define as *search drivers*. search driver

[1] Even though SGP does not involve alternative evaluation functions, it allows the replacement of a problem-specific objective with a metric that, e.g., enables more efficient search operators (cf. [143]). By this token, it is also included in this table.

Table 9.1: Summary of key properties of the approaches presented in Chaps. 4–7. GP: genetic programming equipped with conventional objective function. IFS: Implicit fitness sharing. DOC: discovery of underlying objectives by clustering. SGP: Semantic GP. PANGEA: Pattern-Guided Program Synthesis.

Method	Chap./ Sect.	Evaluation function	Part of execution record used	Objective	Source
GP	1	f_o (1.7)	Outcome vector	Yes	
IFS	4.2	f_{IFS} (4.4)	Outcome vector	No	[124]
Cosolvability	4.3	f_{CS} (4.7)	Outcome vector	No	[94]
DOC	4.4	f^j_{DOC} (4.11)	Outcome vector	No	[95]
SGP	5	d (5.3)	Program semantics	Yes	[135]
Trace consistency	6	f_{TC} (6.4)	Traces (equivalence)	Yes	[100]
PANGEA	7	f_e, f_c (7.1,7.2)	Traces (content)	Yes	[101]

As we argue in the following, search drivers are not required to objectively assess candidate solutions in the context of given problem – this is what the correctness predicate (1.1) is for. They are to *guide* search, meant as an iterative improvement process, by creating a gradient toward better performing solutions. The concept of search driver is thus a generalization of evaluation function.

To get a better grip on this new concept, let us recall first that objective function is not necessarily the best tool to guide a search (Sect. 2.2). On the other hand, none of the evaluation functions in Table 9.1 can be claimed *universally* best, as this would violate the *No Free Lunch Theorem* [193, 192]. Thus, rather than looking for an imaginary 'Holy Grail' of evaluation functions, we ask: can we characterize the minimal set of requirements which, when met by a given evaluation function, make it a useful (even marginally useful) tool to guide search? What is the minimal amount of feedback an evaluation function must provide so that we may talk about any guidance at all? Answering these questions in a rigorous manner will help us delineate search drivers.

No Free Lunch Theorem

9.2 Design rationale

Heuristic search algorithms alternate two actions: generation of new candidate solutions (variation) and their selection (cf. Fig. 1.1):

$$\ldots \xrightarrow{o} P_i \xrightarrow{sel} P' \xrightarrow{o} P_{i+1} \xrightarrow{sel} \ldots, \tag{9.1}$$

where $P_i \subset \mathcal{P}$ is the state of population in the ith iteration of search, and $P' \subset P_i$ is the sample (a multiset in general) of candidate solutions selected from P_i by the selection phase sel. The candidate solutions in P' form the basis for creating new candidate solutions with o, which may internally implement several specific search operators. The new candidate solutions populate P_{i+1}.

Formula (9.1) is a succinct phrasing of an Evolutionary Algorithm (EA) and virtually any iterative heuristic search algorithm. For instance, in a $\mu + \lambda$ evolutionary strategy [151], $|P_i| = \mu + \lambda$, and sel returns the μ best parents from P_i, which are then augmented by λ offspring by o, leading so to P_{i+1}. In local search, $P' = \{p\}$ and P_i is a sample of p's neighborhood. In an exhaustive local search, o generates the entire neighborhood P_{i+1} from the current candidate solution p, and sel selects the best candidate from it. In stochastic local search, o generates only a sample of neighborhood. Even random walk conforms to (9.1): in that case, o generates a random candidate solution, and sel is an identity function.[2]

Search operators in o may also utilize the evaluation outcomes, though in conventional EAs it is usually not the case, and in the following we will be agnostic about that aspect. For clarity, we also ignore for now initialization and termination (Fig. 1.1).

Evaluation is not explicitly present in (9.1). We assume that sel is 'informed' by some evaluation process. Conventionally, an evaluation function f is globally defined, static, and absolute, like the objective function f_o. The concept of search driver originates in observations that *these properties can be substantially relaxed*:

Observation 1: The evaluations $f(p)$ that guide sel's decisions do not need to be independent across the candidate solutions $p \in P_i$. $f(p)$ may change subject to removal of any other $p' \in P_i$. This is the case when evaluation is contextual, e.g., for IFS, CS, and DOC in Chap. 4. In such cases, f's domain is effectively $\mathcal{P}^n$, where n is population size, and it returns an n-tuple of evaluations of its arguments.

Observation 2: The implementation of sel usually involves multiple invocations of a selection operator, each returning a single candidate solution from P_i picked from a relatively small sample $P'' \in P_i$ of candidates, while the characteristics of the remaining solutions in $P_i \setminus P''$ are irrelevant. At that particular moment, f does not even have to be defined outside of P''. For instance, tournament selection cares only about the evaluations of a few candidate solutions that engage in the tournament.

[2] Formula (9.1) is actually expressive enough to embrace *any* search algorithm, including exact algorithms like A*, once we assume that P_i, rather than being a subset of $\mathcal{P}$, is a more general *state* of search process – a formal object that stores the knowledge about the search conducted to a given point, like the prioritized queue of states in A*.

Observation 3: For selection, it is sufficient to make *qualitative* comparisons on f for the candidate solutions in P_i (or in a sample $P'' \in P_i$ – cf. Observation 2). The absolute values of f are irrelevant; it is *orders* that matter. Tournament selection is again a good example here. Moreover, comparisons between candidate solutions do not necessarily need to conform to the requirements of completer orders (in particular to transitivity). Also, some candidate solutions can be arguably incomparable. Algorithms that conduct search only by qualitative comparisons are sometimes referred to as *comparison-based algorithms* [179].

These characteristics form the minimal set of properties of search drivers posited in the beginning of Sect. 9.1. They are deliberately very modest: a typical evaluation function, like the objective function f_o, imposes a complete ordering on *all* candidate solutions in $\mathcal{P}$ and thus conforms Observations 1–3 by definition. Apart from these properties, a typical evaluation function has usually other characteristics that are not essential for search algorithm as defined above, and are in this sense redundant. The chasm between the characteristics of common evaluation functions and the actual needs of a selection operator in a search process (9.1) as expressed in Observations 1–3 motivate the concept of search driver presented in the next section. For clarity, our further argument will primarily focus on individual candidate solutions and local search algorithms. However, we will also present how this considerations generalize to population-based algorithms.

9.3 Definition

search driver

A *search driver* for a solution space $\mathcal{P}$ is any non-constant function

$$h : \mathcal{P}^k \to \mathbb{O}^k, \tag{9.2}$$

where $k \geq 1$ is the *order* of the search driver driver, and $\mathbb{O}$ is a partially ordered set with an outranking relation $\prec$ (precedence)[3]. When applied to a k-tuple P of candidate solutions from $\mathcal{P}$, $h(p)$ returns a k-tuple of evaluations (*scores*) from $\mathbb{O}$. We assume that the scores $o_i \in h(P)$ correspond one-to-one to the arguments $p_i \in P$ and are invariant under permutations of the arguments in P. Accordingly, we allow for abuse of notation and write $h(P)$ even when P is not a tuple but a set. In the following, $o_1 \prec o_2$ means that o_2 is more desirable than o_1. One may alternatively say that $h(P)$ returns a partially ordered set (*poset*) $(P, \prec)$.

behavioral search driver

Many of the search drivers considered in this book are behavioral. A behavioral search driver can be always implemented with help of execution record, because an execution record is the complete account of program execution

[3] $\mathbb{O}$ should not be confused with $\mathcal{O}$, the type of program output (Sect. 1.1).

(Sect. 3.1). In this light, such search drivers could be redefined as mappings from a domain of execution records $\mathcal{E}$ to $\mathbb{O}$; however, to embrace also the non-behavioral search drivers, in the following we conform to the signature in (9.2).

A search drivers with completely ordered codomain $\mathbb{O}$ will be referred to as *complete search drivers*. The conventional scalar real- or integer-valued evaluation functions are examples of such drivers. The motivation for using partial orders is to allow search drivers to abstain from deeming some solutions better than others, which may be desirable when two solutions fundamentally differ in characteristics. As complete orders are special cases of partial orders, an evaluation function with the signature (9.2) that orders its arguments linearly is a search driver as well. On the other hand, search drivers generalize evaluation functions.

complete search driver

A search driver is *context-free* if the scores it assigns to its arguments are independent, i.e. it can be expressed by a one-argument function $f : \mathcal{P} \to \mathbb{O}$:

context-free search driver

$$h(p_1, \ldots, p_k) = (f(p_1), \ldots, f(p_k)). \tag{9.3}$$

Most of conventional evaluation functions f used in EC and GP are context-free search drivers in this sense. Note that for context-free search drivers, the order k is irrelevant. A search driver that is not context-free will be referred to as *contextual*. Note that the signature given in (9.2) is necessary to correctly define a contextual search driver: for instance, when defining $f_{\text{IFS}}(p)$ (4.4) we silently allowed for an abuse of notation, as $f_{\text{IFS}}(p)$ depends not only on p, but also on the other members of the population that p belongs to (the context).

contextual search driver

This definition of search driver follows the design rationale presented in the previous section. Firstly, h evaluates a *set* of candidate solutions (or more precisely a tuple) rather than individual solutions, and is thus contextual (Observation 1). A candidate solution p may receive different evaluations depending on the remaining elements of P, the argument of $h(P)$. Formally, $h(\ldots, p_i, \ldots) = (\ldots, o_i, \ldots)$ and $h(\ldots, p_j, \ldots) = (\ldots, o_j, \ldots)$ where $p_i = p_j$ does not imply that $o_i = o_j$, because the partial orders returned by h in these two applications can be in general completely unrelated. Secondly, the characteristics of the remaining candidate solutions in $\mathcal{P}$ do not have to be known (nor even computable) at the moment of applying h to a particular subset of them (Observation 2). Thirdly, the evaluations assigned to particular elements of P are not only allowed to be qualitative, but also only partially ordered (Observation 3).

Our rationale behind naming this formal object 'search driver' is twofold. The word 'search' signals that the class of problems we are primarily interested in here are search problems, even if they are disguised as optimization problems in GP. The 'driver' is to suggest that, other than creating *some* search gradient, a search driver is not promising to necessarily reach the search target (or detect the arrival at it) – in contrast to what the term 'objective function' suggests. In this sense, search drivers care more about *evolvability* than about reaching the ultimate goal of search.

The contextual evaluation functions discussed in Chap. 4 (IFS, CS, and DOC) can be phrased as complete search drivers, which we illustrate this with the following example.

Example 9.1. Refer to Example 4.1 of IFS evaluation and Table 4.1. The f_{IFS} evaluation function applied there to population $P = (p_1, p_2, p_3)$ produces $f_{\text{IFS}}(p_1) = 2$, $f_{\text{IFS}}(p_2) = 3/2$, and $f_{\text{IFS}}(p_3) = 1$ as the corresponding evaluations. An equivalent order-3 search driver h can be defined as

$$h(p_1, p_2, p_3) = (2, 3/2, 1). \tag{9.4}$$

More generally,

$$h(p_1, p_2, p_3) = (a, b, c), \tag{9.5}$$

where $a, b, c \in \mathbb{O}$ could be abstract values ordered as follows: $c \prec b \prec a$.

Assume that we consider the differences on f_{IFS} for the pairs (p_1, p_2) and (p_2, p_3) too small to deem any of the compared candidate solutions better. In such a case, one could redefine the order in $\mathbb{O}$ so that so that only $c \prec a$ would hold.

Last but not least, the order of a search driver does not have to be bound with the size of population; an exemplary order-2 search driver for this problem could be defined as

$$h(p_1, p_2) = h(p_2, p_3) = h(p_1, p_3) = (a, b), \tag{9.6}$$

where $a \prec b$. ■

9.4 Search drivers vs. selection operators

A vigilant reader might have noticed that search drivers can be directly used as selection operators. Indeed, $h(p_1, \ldots, p_k)$ is a k-tuple of values $(o_1, \ldots, o_k)$ from $\mathbb{O}$, and the maximal elements[4] in $(o_1, \ldots, o_k)$ are obvious candidates for being selected. Should $\mathbb{O}$ be only partially ordered, the selection between incomparable elements in $\mathbb{O}$ could be addressed by random drawing.

Though these similarities might suggest equivalence of search drivers and selection operators, we find it important to distinguish between these two

[4] In general, the maximal elements in the sense of partial orders. A partially ordered set may have arbitrary many maximal elements.

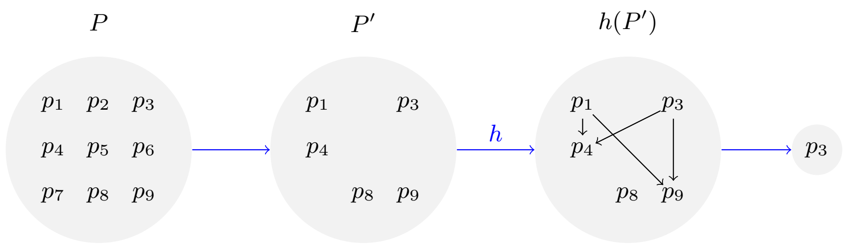
P
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9
P'
p_1 p_3 p_4 p_8 p_9
h
$h(P')$
p_1 p_3 p_4 p_8 p_9
p_3

Fig. 9.1: The process of selection involving a single search driver. The selection operator draws a sample P' of candidate solutions from the population P. Search driver h is applied to P' and returns a partial ordering of the elements of P', in which it may deem some pairs of candidate solutions incomparable (e.g., p_1 and p_3). Finally, the selection operator makes a decision about selection based on h, yielding p_3.

concepts. A selection operator is a component of metaheuristic architecture (9.1) that implements the *entirety* of preferences concerning navigation in the search space as well as the desired characteristics of solutions. A single search driver, to the contrary, is an *elementary* source of information, reflecting only selected characteristics of candidate solutions. This distinction becomes particularly clear when using multiple search drivers simultaneously (Sect. 9.8). Thus, selection should be seen as a higher-level component that may engage one or more search drivers, applying them to accordingly prepared (typically drawn at random) samples of candidate solutions. We illustrate this principle in Fig. 9.1.

Search drivers differ from selection operators also in scope of application. A selection operator is typically applied to entire working population. The scope of search driver is determined by its order k, which we assume to be usually low compared to population size (recall that in Sect. 9.1 we set out to define the *minimal* feedback needed to effectively guide search).

Another difference is determinism. We define search drivers as deterministic functions, leaving the non-deterministic aspect of selection to a selection operator. In particular, we assume that it is the selection operator that is responsible for drawing a randomized sample of candidates, which are then passed to a search driver. Handling special cases with randomness (e.g., tie-breaking) is also delegated to selection operator in our conceptual model.

9.5 Universal search drivers

Remarkably, our definition of search driver in (9.2) does not refer to a program synthesis task. A search driver defines certain features of a solution, whether in the context of a specific program synthesis task or in a more abstract way. This is not incidental: we intend to embrace also the *universal search drivers* that promote problem-independent characteristics of candidate solutions. Examples of universal search drivers for program synthesis include, but are not limited to:

universal search driver

- Non-functional properties of programs like *program length* (size), *execution time*, *memory occupancy*, or *power consumption.*
- *Input sensitivity.* In cases where program input is a tuple of variables, it might be important to synthesize programs that take them all into account. There are two variants of this characteristic. A program can be said to be *syntactically sensitive* to all variables if it *fetches* them all (reads them in). A program is *operationally sensitive* if it can be shown that for every variable v_i there exists a combination of the remaining variables such that the output of a program changes when v_i is being changed. A syntactically sensitive program does not have to be operationally sensitive. The latter property is usually more desirable, but the former one is easier to verify.
- *Evolvability.* Given the iterative nature of search process, it is desirable to promote candidate solutions that can be subsequently modified to make further variation possible, and help reaching the optimal solution in a longer perspective. In tree-based GP, evolvability is hampered by, among others, bloat: standard search operators tend to extend deeper parts of programs (close to program leaves), while such changes often have no impact on the behavior of a program.
- *Smoothness.* In symbolic regression, programs are real-valued functions that are often required to be smooth, i.e. the output of a program should not change abruptly in response to modifications of the input.

Note that most of the above properties are behavioral, which is in tune with the leading motif of this book. Also, some of them are inherently qualitative; for instance, syntactic input sensitivity is basically a binary property. Some other characteristics are more quantitative but naturally deserve approximate comparisons. For instance, minor differences of program length are negligible in most applications: it really does not matter whether a symbolic regression model has e.g., 38 or 39 instructions. Another scenario that calls for tolerance is when the underlying measure is noisy. These examples show that defining search drivers in a qualitative manner is practical.

Example 9.2. An order-2 search driver that maps the exact program length onto a qualitative indicator can be defined as

$$h(p_1,p_2) = \begin{cases} (0,1), & if |p_1| > |p_2| + \beta \\ (1,0), & if |p_1| < |p_2| - \beta \\ (0,0), & otherwise \end{cases}, \tag{9.7}$$

where $|p|$ is the length of program p, β is the tolerance threshold, and $0 \prec 1$. This search driver renders any two programs that differ in length by less than β as indiscernible, and so imposes a complete order (pre-order to be precise) on its arguments. Note that the outranking relation defined by h is in this case intransitive.

Should it be more appropriate to use partial orders, the codomain could be extended to $\mathbb{O} = \{0, 1, \phi\}$, where $0 \prec 1$would be the only outranking in $\mathbb{O}$. Then, the third case in (9.7) would return (ϕ, ϕ), so that minor differences in program length would be interpreted as incomparability. ■

9.6 Problem-specific search drivers

A problem-specific search driver is a search driver that refers to the specification of program synthesis task. The k-tuples returned by such a driver depend not only on the k arguments (programs) but also on the *Correct* predicate of a task of consideration (Sect. 1.2). We assume the dependency on the latter to be implicit, unless otherwise stated.

The conventional objective function f_o may be trivially cast as the following complete, problem-specific, context-free search driver of an arbitrary order k:

$$h(p_1, \ldots, p_k) = (f_o(p_1), \ldots, f_o(p_k)). \tag{9.8}$$

By (1.7), such a search driver depends on T, i.e. all tests that define a given program synthesis task. Alternatively, we may consider a search driver that counts program's failures in an arbitrary *subset* of tests $T' \subset T$. For instance, T' could have been determined by an underlying objective (4.10) in the DOC method presented in Sect. 4.4. A more sophisticated variant is an order-2 search driver based on the inclusion of passed tests

$$h(p_1, p_2) = \begin{cases} (0, 1), & if\, T(p_1) \subset T(p_2), \\ (\phi, \phi), & otherwise \end{cases} \tag{9.9}$$

where we recall that $T(p) \subseteq T$ is the subset of tests passed by program p, and $\phi \in \mathbb{O}$ is the special incomparable value (cf. Example 9.2). Such h defines a partial order that implements the concept of *measure* on a set. An extreme case is a search driver that depends on programs' outcomes on a individual tests in T, i.e. dominance on tests

$$h(p_1, \ldots, p_k) = (g(p_i, t), \ldots, g(p_k, t)), \tag{9.10}$$

where we recall that $g : \mathcal{P} \times \mathcal{T}$ is an interaction function (see 4.1).

The repertoire of problem-specific search drivers is by no means exhausted by the above examples. Virtually any evaluation function defined in past studies can be recast as a search driver in an analogous way. On the other hand, search drivers themselves represent a large class of functions. This helps convey that, in a sense, there is nothing special about the conventional test-counting objective function f_o: it is just one of many possible search drivers, but not necessarily the most effective one for a given program synthesis task (or a class of tasks). This observation leads to questions on quality of search drivers, which we address in the next section.

9.7 Quality of search drivers

We are ultimately interested in designing search drivers that perform well. However, what does 'perform' mean for a search driver? A search driver is just one component of iterative search algorithm, hidden inside the selection step in (9.1). The overall performance of the algorithm depends not only on the engaged search driver(s), but also on the selection operators, search operators, population initialization method, and possibly other components.

This suggests that it might be difficult, if not impossible, to investigate the the quality of a search driver in isolation from the remaining components of an iterative metaheuristic search algorithm[5]. Therefore, in this book we assume that the best gauge of a search driver's usefulness is its empirical performance on actual problem instances. In accordance with this, we introduce three categories of search drivers *for a given class of iterative search algorithms A* (9.1) and a set of problems (e.g., a suite of benchmarks). To this end, we define first the concept of random search driver.

random search driver

An order-k *random search driver h* is a search driver such that, for any given P and $(\dots, o_i, \dots, o_j, \dots) = h(P)$ it holds

$$\Pr(o_i \prec o_j) = \Pr(o_j \prec o_i), \tag{9.11}$$

that is, it is equally likely that h orders o_i before o_j and that it orders them reversely. Note that in general $\Pr(o_i < o_j) \leq 1/2$ due to the potential presence of incomparability.

The random search driver serves as a reference point for defining *effective*, *deceptive*, and *neutral* search drivers:

effective search driver

- An order-k search driver is *effective* if it reduces the expected number of iterations of A compared to the number of iterations of A equipped with an order-k random search driver.

[5] In the process of writing this book, we have undertaken an attempt of designing a formal, algorithm-independent measure of search driver quality, based on the concordance of solution orderings provided by search drivers with the first hitting times of solutions. Unfortunately, the formalism required making many unrealistic assumptions, so we decided to not present it here.

- An order-k search driver is *deceptive* if it increases the number of iterations mentioned above. deceptive search driver
- An order-k search driver is *neutral* if it does not affect the number of iterations in a statistically significant way. neutral search driver

We are obviously interested in effective search drivers, and with this book hope to pave the way for practical development and principled design thereof. From now on, by ‘search driver’ we will mean an effective search driver, unless otherwise stated.

We intentionally attribute deception to search drivers rather than to problems, even though the latter is prevailing in the literature. Recall that program synthesis tasks we consider here are inherently search problems that are only disguised as optimization problems (Sect. 1.5.3). A search problem cannot be deceptive, because all it defines is a set of candidate solutions (states) and the goal predicate ($Correct$ in program synthesis). This the true, underlying information about the problem; no suggestion is being made that some of the candidate solutions are ‘closer’ to the search goal than others (whatever ‘closer’ would mean in this context). It is only a search driver (or an objective function in the more conventional setting) that can be deceptive in the above sense.

An *optimal order-k search driver* is a search driver h^* that maximizes, in the above sense, the performance of a given search algorithm on a given set of problems. This concept will serve us as a useful reference point in the following. optimal search driver

9.8 Employing multiple search drivers

Effective search drivers can be informally divided into *weak* and *strong* depending on how much they reduce the expected number of iterations required to reach the correct program. However, even weak drivers are assumed to perform significantly better than the random ones in the above sense. weak search driver strong search driver

Designing strong search drivers for program synthesis is difficult; if it was not, program synthesis would be a solved problem. In contrast, weak search drivers are by definition poor guides for a search process. Indeed, the studies we conducted earlier [97, 101] suggest that many search drivers do not work particularly well in isolation. However, given that search drivers may reflect different qualities of candidate solutions (see examples in Sects. 9.5 and 9.6), it is natural to consider using many of them.

There are many ways in which the judgments of multiple search drivers can be translated into decisions made by selection operator (or, in a wider context, the behavior of all algorithm components). We divide the techniques that facilitate usage of multiple search drivers into *sequential* and *parallel*. The sequential methods allow different search drivers to be used at various stages of an iterative search process. In the simplest realization, the choice of search driver is explicitly controlled by the method. For instance, an analogous idea of alternating multiple evaluation functions (together with problem instances) at regular time intervals has been exploited to induce modularity in evolved neural networks [74, 75]. Interestingly, such proceeding can be seen as yet another form of *shaping* which we touched upon in Sect. 4.2 [175].

sequential usage of search drivers

shaping

Explicit control of the choice of search driver requires multiple design choices: in which order they should be used, for how many iterations, and whether they should take turns cyclically. Also, the transitions between particular search drivers suddenly change the 'rules of the game': adaptations acquired under one search driver may turn out to be maladaptations under subsequent search driver. Such 'catastrophic events' can be interesting when trying to reproduce some biological phenomena in silico (like in [74, 75]), but are not necessarily useful in program synthesis.

parallel usage of search drivers

In this book, we argue for using multiple search drivers *in parallel*, primarily because this is consistent with our stance that no single search driver is a perfect means to control a search process, and thus no search driver should be favored. Another motivation is the above-mentioned problematic parameterization of sequential techniques. But even more importantly, parallel usage of search drivers has other appealing features discussed in below.

Partial independence. Assume n search drivers $h_i, i = 1, \ldots, n$. Let us denote by D_i the event that h_i orders two programs p_1 and p_2 discordantly with the optimal search driver h^*, i.e.

$$\Pr(o_1^i \prec^i o_2^i \wedge o_2^* \prec^* o_1^*). \tag{9.12}$$

Given the optimal nature of h^*, such discordance may result in an increase of the expected number of iterations. The probability that any pair of search drivers is simultaneously discordant with h^* is less or equal the probability that any of them is discordant. By the sum rule:

$$\Pr(D_i \cup D_j) = \Pr(D_i) + \Pr(D_j) - \Pr(D_i \cap D_j), \tag{9.13}$$

from which it follows that $\Pr(D_i \cap D_j) \leq \Pr(D_i)$, as by definition $\Pr(D_i \cup D_j) \geq \Pr(D_j)$. Simultaneous availability of h_i and h_j lowers thus the risk of mistakenly deeming one candidate solution less useful than another. The greater the number of search drivers, the less likely it becomes for them to be simultaneously discordant with the optimal search driver, and that likelihood is the lower the more independent are the search drivers in question. In the extreme case of n *fully independent search drivers* , the probability of all of them being simultaneously discordant quickly vanishes with n:

independent search drivers

$$\Pr\left(\bigcap_{i=1}^{n} D_i\right) = \prod_{i=1}^{n} \Pr(D_i). \tag{9.14}$$

On the other hand, the probability of all drivers being simultaneously *concordant* is also decreasing with n. However, if the search drivers in question are effective, they are likely to make more concordant decisions than discordant ones, i.e. $\Pr(o_1^i \prec^i o_2^i) < 1/2$, and the probability of at least half of n drivers to be concordant is greater than $1/2$. For the special case of $\Pr(D_i) = \Pr(D_j), \forall i,j \in [1,n]$, that probability is determined by the cumulative distribution function of binomial distribution. Thus, if partially independent search drivers were to vote about a relation between a pair of candidate solutions, they are more likely to make the right decision than the wrong one.

Although designing a family of independent search drivers may be difficult, a certain degree of independence comes 'for free' for the search drivers summarized in Table 9.1 and examples given in Sects. 9.5 and 9.6, because they peruse different aspects of program behavior. Some methods promote independence on their own; for instance, the particular derived objectives built by DOC (Sect. 4.4) are based on disjoint subsets of tests, and may by that token be partially independent.

The above argument is analogous to the motivations for *committees* of classifiers in machine learning (a.k.a. *classifier ensembles*) [12]. ML committees are usually built to provide more robust predictions in the presence of noisy data. Just as multiple search drivers are less likely to simultaneously commit an error, so for the classifiers that vote about the output (decision class label or continuous signal) to be produced for a given example.

classifier ensembles

More specifically, partial ordering of candidate solutions performed by a search driver can be seen as an ordinal regression task (a partial one, to be more precise). A search driver is thus a special case of regression machine, and as such is characterized by certain *bias* and *variance* [38]. Bias represents a driver's inherent propensity toward certain realizations (models), while variance reflects the variability of a model's predictive accuracy. These quantities are inseparable, a characteristics known as *bias-variance tradeoff*: a highly biased predictor tends to have low variance and vice versa. However, by aggregating multiple low-bias, high-variance predictors, the variance can be reduced at no extra cost to bias. This observation is the key motivation for ensemble machines, and is naturally applicable also to search drivers.

bias-variance tradeoff

Diversity. By using several search drivers of different nature in parallel, we hope to provide for greater behavioral diversity in a population. Promoting behavioral diversity entails genotypic diversity, i.e. diversity of program code in the case of program synthesis. The importance of diversity maintenance has been demonstrated in population-based search and optimization techniques many times in the past, and was the major premise for designing methods like implicit fitness sharing (Sect. 4.2). We find diversity maintenance by means of behavioral search drivers particularly natural, as opposed to, e.g., niching techniques [118] and island models [191] that require parameter tuning.

multi-modality

Multimodality. Program synthesis tasks are often multimodal, i.e. feature multiple optimal solutions, all of them conforming to the correctness predicate. A single-objective search process may be biased in tending to explore only selected basins of attraction of such optimal solutions. A search process that follows multiple objectives in parallel may be more open to explore many such basins. By the same token, multimodality is an argument against sequential usage of search drivers. Consider interlacing two search drivers along iterations of search process; if one of them happens to drive the search toward one optimum while another toward another optimum, search may cyclically oscillate between these optima.

Moderate computational overhead. Different search drivers may share algorithmic components needed to compute them. In such cases, calculating multiple search drivers rather than one does not necessarily incur massive overheads. For instance in TC (Chap. 6) and PANGEA (Chap. 7), recording of a program trace is a side effect of its execution and as such causes only moderate overhead.

9.9 Multiobjective selection with search drivers

In conventional GP, a scalar evaluation function serves as the basis for a straightforward selection operator (e.g., tournament selection). Simultaneous usage of several search drivers argued for in the previous section precludes direct application of such operators and requires special handling. In the following we discuss several alternative means to that end, most of which are only applicable when search drivers are complete, i.e. impose linear (pre)orders on candidate solutions.

Fig. 9.2 presents an example of a selection process involving two search drivers. As in the single-driver case (Fig. 9.1), the role of search drivers is to provide recommendation (a partial order of the considered sample of candidate solutions P'), while selection process is responsible for drawing P' and final selection of the 'winner'. Recommendations of particular drivers may be contradictory: for instance, p_1 and p_3 are incomparable according to h_1, while h_2 suggests that the former is worse than the latter.

The role of selection algorithm is to reconcile such discrepancies and appoint the best candidate solution within the considered sample P'. How to do

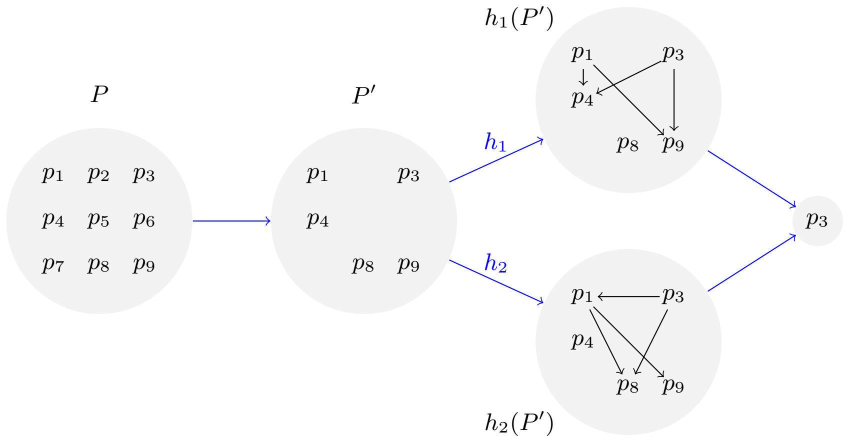

Fig. 9.2: The process of selection involving two search drivers h_1 and h_2. The selection operator has to reconcile the – partially contradicting – orderings provided by h_1 and h_2 to produce the final selection outcome, i.e. p_3. Consult Fig. 9.1 for an analogous single-driver example.

this appropriately and efficiently (particularly when the number of search drivers large) is in itself an interesting research question, which we do not address in this book. In the following, we discuss the possible aggregation methods for the case when all search drivers in question are complete, i.e. (pre)order the candidate solutions linearly.

Aggregation (scalarization). The arguably easiest way of handling multiple search drivers is to merge them into scalar evaluation, to be interpreted later by a single-objective selection method. If the codomains of search drivers in questions happen to be defined on metric scales, this may boil down to applying an averaging operator like arithmetic mean. Another alternative, geometric mean, is equivalent (up to an order) to the concept of *hypervolume* in multiobjective optimization. When search drivers range in different intervals and have different distributions, rank-based aggregation can be used. hypervolume

Scalar aggregation opens the possibility of using numerous conventional selection operators. On the other hand, aggregation incurs compensation (Sect. 2.2.2). Nevertheless, as we showed in [101], even a simple multiplicative aggregation of search drivers can offer substantial performance improvements.

Lexicographic ordering. A common multiobjective selection technique that avoids explicit aggregation is *lexicographic ordering*. This method expects the search drivers to be sorted with respect to decreasing importance. Given two candidate programs p_1 and p_2, they are first compared on the most important driver. If this comparison is conclusive, e.g., p_1 is strictly better than p_2 on that criterion, p_1 is selected. Otherwise, the next driver with lexicographic ordering respect to importance is considered. This process repeats until one of the programs proves better. Should that not happen, p_1 and p_2 are declared indiscernible. In other words, the consecutive drivers resolve the ties on the previous ones.

Lexicographic ordering avoids direct aggregation, but in exchange for that requires domain-specific ordering of search drivers. Also, it becomes effective only for discrete and coarse-grained objectives. If for instance the most important search driver happens to feature many unique values, the remaining search drivers have little to say.

lexicase selection

Lexicase selection. An interesting recent method that builds upon lexicographic ordering is *lexicase selection* [50]. The main difference with respect to lexicographic approach is in the adopted ordering of search drivers which is random and drawn independently in every selection act. This helps avoiding overfocusing on some search drivers and diversifies the population. This straightforward and parameterless method proved very efficient in [50], where it was applied to tests, especially when the number of them was substantial. However, when applied to a moderate number of search drivers, its diversification capability and performance deteriorate [114].

In retrospect, lexicase selection can be seen as a test sampling technique. Such techniques typically draw a random sample of tests $T' \subset T$ in every generation of GP run, and use only the tests from T' for evaluation in that generation. Test sampling fosters diversity and improves performance, even when brought to extremes, i.e. drawing just a single test in each generation [43]. Lexicase selection 'individualizes' this process for particular selection acts.

Multiobjective selection. Multiobjective evolutionary algorithms offer several methods that avoid the pitfalls of aggregation while still eliciting useful information on search gradient. Usually, the underlying formalisms are dominance relation and Pareto ranking. Of the multiobjective selection methods, the Non-dominated Selection Genetic Algorithm (NSGA-II, [26]) is arguably most popular. NSGA-II employs a tournament selection on Pareto ranks to make selection choices. As a tie-breaker, it employs *crowding*, a measure that rewards the candidate solutions that feature less common scores on search drivers. The method is also elitist in selecting from the combined set of parents and offspring, rather than from parents alone. NSGA-II is the selection algorithm used in the experiment reported in Chap. 10. Many past works in GP proved the usefulness of multiobjective approach; see, e.g., [25], where an ad-hoc multiobjective algorithm was used for simultaneous promotion of diversity and reduction of program bloat.

9.10 Related concepts

There are several concepts in computational and artificial intelligence that bear some resemblance to that of a search driver.

In EC, the concept that arguably resembles search driver is *surrogate fitness*. Also known as *approximate fitness function* or *response surface* [71], a surrogate fitness function provides a computationally cheaper approximation of the original objective function. Surrogates are particularly helpful in domains where evaluation is computationally expensive, e.g., when it involves simulation. They usually rely on simplified models of the process being simulated, hence yet another alternative name: *surrogate models*. In continuous optimization, such models are typically implemented using low-order polynomials, Gaussian processes, or artificial neural networks. Occasionally, surrogate models have been also used in GP. For instance, in [51], Hildebrandt and Branke proposed a surrogate fitness for GP applied to job-shop scheduling problems. A metric was defined that reflected the behavioral similarity between programs, more specifically how the programs rank the jobs. Whenever an individual needed to be evaluated, that metric was used to locate its closest neighbor in a database of historical candidate solutions and neighbor's fitness was used as a surrogate.

surrogate fitness

approximate fitness function

response surface

Search drivers diverge from surrogate fitness in several respects. Firstly, surrogate functions are by definition meant to *approximate* the original objective function. Search drivers are, to the contrary, based primarily on the evidence that objective functions are not always the best means to navigate in a search space. Given the deficiencies discussed in Sect. 2.1 and the experimental evidence backing up the methods presented in Chaps. 4–7, why would one insist on approximating an objective function? Secondly, search drivers are primarily intended for search problems rather than optimization problems. This leaves more freedom in their design, which do not have to 'mimic' an objective function across the entire search space. Thirdly, in a program synthesis task, a search driver is not required to be consistent with a correctness predicate (1.5). In surrogate fitness, such consistency is essential.

Augmenting search with additional objectives is a part of the methodology proposed in [76] under the name of *multiobjectivization*. The additional objectives are introduced in that framework to make search more efficient, turn the original single-objective problem into a multiobjective one, and solve it using more or less off-the-shelf algorithms capable of handling multiobjective problems. An important assumption is that the extra objectives convey some *additional* problem-specific knowledge. In contrast, many of the search drivers discussed here essentially 'rephrase' the information that is conveyed – albeit subject to losses discussed in Chap. 2 – by the conventional objective function. The decomposition of scalar evaluation into multiple objectives in DOC (Sect. 4.4) is an example of such a proceeding. Also, with search drivers we put more emphasis on having many, even qualitative, information sources.

multiobjectivization

The concept that is close to both multiobjectivization and search drivers is that of *helper objectives* [70], additional objectives used along with the original ('primary') objective in multiobjective setting. According to the cited

helper objectives

work, helper objectives are meant to maintain diversity in the population, guide the search away from local optima, and help creation of good building blocks, meant as small components (*schemata*, to be more precise) that positively contribute to solution's evaluation [41]. The author argues that they should be 'in conflict' with the original objective function; for search drivers, we formalized a related concept of partial independence in Sect. 9.8. Helper objectives may change with time, i.e. only a subset of helper objectives is being used at any given time (dynamic helper objectives). The cited work also addresses the issue of size of Pareto front in multiobjective setting: once the number of candidate solutions with the same values of objectives exceeds the *niche count*, candidate solutions are randomly removed so that this constraint is not violated anymore. The approach was applied to job-shop scheduling and traveling salesperson problems, producing encouraging results.

Search drivers diverge from helper objectives in several respects. Helper objectives are meant to be used primarily with optimization problems – that is why the original objective function is always included as one of the objectives. Search drivers address search problems, in particular program synthesis, where achieving the sought solution is verified with a separate correctness predicate, so discarding the original objective function is acceptable. Also, designing helper objectives is similar to multiobjectivization in requiring substantial domain knowledge: for instance in application it to job-shop scheduling in [70], helper objectives reflected the flow-times of particular jobs. Later work also concerns job-shop scheduling [116] and confirms this deep immersion in problem domain. Search drivers, to the contrary, are more generic, and as we showed in Sect. 9.5, many of them are universal. Finally, search drivers are more general than helper objectives in being allowed to impose only qualitative and partial relationships between candidate solutions (see also the summary of search drivers' properties in Sect. 9.12).

In GP, several approaches have been proposed that aim at reducing the number of tests used for evaluation of candidate programs. In doing that, such methods effectively replace the original objective with an alternative evaluation function that can be sometimes likened to a search driver. The probably oldest contribution in this category is dynamic training subset selection [37]. In [115], selection of tests was applied to the task of software quality classification, in an attempt to reduce overfitting. In [43], this has been taken even further, i.e. single tests have been used. Several variants of this approach studied in [42] consistently reduced overfitting compared to standard GP.

heuristic function

In AI, the concept that bears certain similarity to search driver is that of *heuristic function*. Heuristic functions in algorithms like A* bound the actual cost of reaching the search goal. Thy can be used to prioritize search and often thereby lead to performance improvements. They can be designed by relaxing the original problem, for instance, for an 8-puzzle problem, this can be achieved by assuming that any two neighboring tiles can be swapped, and counting the number of moves so defined. Interestingly, methods exist that generate heuristic functions automatically given problem formulation [156].

In reordering the visiting of candidate solutions, heuristic functions indeed resemble search drivers that may lead search in different directions than that of the objective function. However, heuristic functions explicitly rely on additional domain-specific knowledge that search drivers do without; examples include straight-line distance between cities in the famous Romanian roadmap example in [156] or the cost function for the 8-puzzle example mentioned above. Also, search driver is a more general formalism than heuristic function: unlike the latter, it is not guaranteed to bound the original objective function. As a consequence, algorithms that rely on search drivers cannot enjoy the 'comfort' of A*, which provably reaches an optimal solution (goal state) provided it exists in the searched tree. Providing analogous bounds in program synthesis is difficult due to the complex genotype-phenotype mapping (Sect. 1.4). In the typical tasks approached with the A*-like algorithms, the effects of search moves on the objective function are well understood, which facilitates designing efficient admissible heuristic functions. In program synthesis, a single move may change program's behavior almost arbitrarily.

reinforcement learning

In *reinforcement learning* (RL, [173]), the concepts of *intrinsic rewards* (or *intrinsic motivation*) [8] and internal rewards [169] bear distant similarity to search drivers. In nontrivial RL tasks, an agent often lacks external 'incentives' to explore and learn from an environment. Both methodologies mentioned above assume that, in the absence of such incentives, an agent would on its own 'induce' appropriate internal/intrinsic motivations and follow them, escaping thereby the detrimental state of temporal 'apathy'. In a sense, intrinsic rewards and intrinsic motivations can be seen as an agents's emanation of curiosity.

novelty search

open-ended evolution

Last but not least, behavior search drivers share motivations with *novelty search* [110] (NS). In the spirit of *open-ended evolution*, often considered in artificial life community, novelty search discards search objectives altogether and rewards individuals for being behaviorally different from their peers in the current population and selected representatives of the search history. This turns out to work well on deceptive problems, especially on the problems where the mapping from genotypes to behaviors is strongly many-to-one, and the resulting behavioral space is relatively small. In [111], the authors applied novelty search to nontrivial GP benchmarks of maze navigation and Artificial Ant. The algorithm diverges from the traditional GP only in evaluation function. Prior to running evolution, an empty, unlimited-capacity archive A is created. Each evaluated individual has a low probability of being included in the archive. The evaluation of an individual in population P is defined as

$$f_{\text{NS}}(p) = \frac{1}{k}\sum_{i=1}^{k} dist(p, \mu_i), \tag{9.15}$$

where $dist()$ is a behavioral distance measure, μ_i is the ith closest neighbor of p in $P \cup A$ with respect to $dist()$, and k defines the size of the neighborhood.

They key component of NS is the behavioral distance measure $dist()$. In [111], it was based on the final location in the maze problem and the time-wise distribution of food collections in the Artificial Ant problem. The authors of [121], the probably first attempt to apply novelty search to generic GP tasks, based $dist()$ on behavioral descriptors closely resembling outcome vectors defined in this book (2.3). The ith vector element is 1 if the individual belongs to an assumed (low) percentile of programs that commit the smallest error on a test; otherwise it is set to 0. Experimental assessment of this configuration on three benchmarks did not yield particularly conclusive results. Other works related to NS include, among others, [196] and [195], where diversification was promoted by maximizing correlation distance between time-wise behavior of GP programs representing trading strategies.

In general, we anticipate NS to struggle when faced with more demanding program synthesis tasks because of the size of behavioral space. Even in the simplest case when tests can be only passed or failed, the number of all possible behavioral descriptors grows exponentially with the number of tests, and becomes staggering even for the small benchmarks typically considered in GP (e.g., 2^{64} for the humble 6-bit multiplexer; see Chap. 10). One may doubt if simply enticing an evolving population to spread across such a space is sufficient to locate the target solution.

Nevertheless, NS shares motivations with behavioral program synthesis: Lehman and Stanley state that “The problem is not in the search algorithm itself but in how the search is guided” [111, p. 842], which strongly resonates with the arguments in this book. NS can be seen as the opposite extreme to conventional search driven by the conventional objective function. Search drivers sit in between these two extremes: they impose different search gradients than the conventional objective function, yet, contrary to NS, those gradients are not entirely detached from it.

9.11 Efficiency

Abandoning search objectives in favor of search drivers has measurable consequences for more technical aspects of implementation. Below we discuss such implications for the arguably most important technical aspect of program synthesis – computation cost. In a typical GP run, the lion’s share of computation is spent on running programs, i.e. applying them to tests. No wonder the number of evaluated programs is the most common unit of computational expense in GP.

In Sect. 9.8, we argued for using several search drivers in parallel, in a multiobjective setting. As some search drivers incur substantial overheads (e.g., classifier induction in PANGEA), this may seem computationally prohibitive. However, the execution record stores the complete account of program execution for a considered set of tests. Once it has been computed for a given program (which is not particularly expensive as demonstrated in Sect. 3.2) most search drivers can be calculated from it at a moderate cost, because the data they require (outcome vectors, program semantics, program traces – see Table 9.1) can be immediately retrieved from the record (Fig. 3.2). Therefore, if the number of tests is large or/and program execution is in itself expensive, the total overhead resulting from using multiple search drivers may become negligible.

Moreover, in some scenarios search drivers allow for substantial *reduction* of computational expense. For instance, the convergence of execution traces in Chap. 6 can be used to reduce the time spent on program execution: if two traces merge (which the method has to detect anyway), i.e. program execution leads to the same execution state for two tests, then from that state on only one execution has to be conducted, as the other must proceed in exactly the same way.

Opportunities for potential savings wait to be uncovered not only in the internals of program execution, but also on the higher abstraction levels. Consider an order-2 search driver h that completely orders the programs according to the number of tests they pass. This search driver can be trivially expressed using the objective function f_o: $h : \mathcal{P}^2 \to \{0,1\}^2$, where $0 \prec 1$ and

$$h(p_1, p_2) = \begin{cases} (0,1), & \text{if } f_o(p_1) > f_o(p_2) \\ (1,0), & \text{if } f_o(p_1) < f_o(p_2) \\ (0,0), & \textit{otherwise} \end{cases} \tag{9.16}$$

(recall that f_o is minimized). This formulation assumes that $f_o(p_1)$ and $f_o(p_2)$ are calculated independently, which for a set of tests T requires the total of $2|T|$ executions of p_1 and p_2. Now consider an algorithm that iterates over T and applies p_1 and p_2 to a given $t \in T$ simultaneously. Let $i \in [1, |T|]$ be the number (index) of the currently processed test, and $f_o^{(i)}(p)$ the number of tests failed so far. Note that as soon as the following condition starts to hold

$$|f_o^{(i)}(p_1) - f_o^{(i)}(p_2)| > |T| - i, \tag{9.17}$$

the loop over i can be terminated, because the outcomes for the remaining $|T|-i$ tests cannot compensate the already gathered evidence in favor of one of the programs. In such a scenario, the total number of program executions amounts to $2i$, and may be substantially smaller than $2|T|$ above. Though the actual benefits resulting from enhancements like this one are domain- and problem-dependent, even a minor reduction of the number of program executions may be beneficial in challenging program synthesis tasks.

9.12 Summary

This chapter described a preliminary attempt to crystallize the concept of search driver, a generalization of evaluation function intended to meet the needs of metaheuristic search algorithms and program synthesis in particular. Further effort is clearly required to get a better grip on it and possibly lead to principled design of search drivers. Nevertheless, it should be clear already at this point that search drivers exhibit common characteristics (cf. Observations formulated in Sect. 9.2):

1. Search drivers are contextual. The role of a search driver is to provide gradient within a relatively small set of candidate solutions. A search driver is not required to provide such a gradient globally.
2. Search drivers provide qualitative, ordinal feedback. Absolute values are irrelevant. What matters is the (partial or complete) order of candidate solutions.
3. Search drivers do not have to relate to the original objective function, and in particular do not have to correlate with it. Preferably, they should approximate the optimal search driver.
4. Search drivers do not have to be consistent in the sense of (1.5), i.e. to indicate the optimality of candidate solutions by achieving extreme values at them (nor in any other way). This functionality is delegated to a correctness predicate.
5. Search drivers may depend on the entire state of the search process meant as the working population of candidate solutions (or even state in a broader sense, for instance including candidate solutions visited in previous iterations).
6. As a consequence of (5), search drivers may be non-stationary, i.e. order the same subset of candidate solutions differently in particular iterations of a search loop.
7. Search drivers can be weak, i.e. order relatively many candidate solutions differently than an optimal search driver. Using many weak drivers in parallel can make the search process effective by providing sufficiently strong search gradient *and* diversity.