Implications of the behavioral perspective

Previous chapters presented a range of approaches that characterize program behavior in terms of execution record and search drivers. The experiments reported in Chap. 10 demonstrated that these approaches increase the likelihood of synthesizing a correct program. What are the other, not necessarily empirical, implications of behavioral program synthesis? We discuss them in a broader context in this chapter.

11.1 Conceptual consequences

When discussing advantages of behavioral evaluation in Sect. 9.8 and elsewhere, we suggested that it can be a means for assessing and promoting diversity in populations of programs. This is particularly natural in the presence of multiple search drivers. If neither of two compared programs dominates the other, selection renders them incomparable and allows them co-exist in a population. No additional mechanism for controlling or inducing diversity may be necessary. Behavioral evaluation and selection implicitly provide for phenotypic diversity, which in turn may lower the risk of premature convergence and overfocusing on local optima.

One might argue that maintaining behavioral diversity is not a truly novel feature, especially given the recent works addressing semantic diversity (e.g., [35]) or methods like lexicase selection [50]. However, the concept of an execution record invites a deeper take on diversity, not limited only to inspecting program output. For instance, two programs could be treated as behaviorally distinct if there is any difference in their execution records. Because program fragments are being constantly moved by crossover between individuals in the population, such a mechanism could promote ‘internal behavioral diversity’ and have positive impact on search performance. This supposition remains to be verified in further studies.
Many of the search drivers considered in this book are in a sense *invented* by a search algorithm. By relying on ML-induced behavioral models, PANGEA can autonomously detect behavioral patterns that reveal potentially useful candidate solutions and parts thereof. No background knowledge or human ingenuity is necessary for that purpose: the experimenter is not required to specify which types of patterns are desirable. This makes PANGEA attractive when compared to, e.g., extensions of reinforcement learning methods (Sect. 9.10) that require the additional search drivers to be explicitly provided [8, 165].

Another, potentially more consequential feature of behavioral evaluation is facilitation of *problem decomposition*. Problem decomposition, often considered together with *modularity*, has been for long considered an important aspect of intelligent systems, and it remains to be an area of intense research in computational and artificial intelligence [190]. In behavioral program synthesis, there are at least two alternative avenues to problem decomposition.

By holding a trace for every test, execution records open the door to *vertical*, row-wise, test-wise problem decomposition. This capability is essential for geometric semantic GP (Chap. 5), where the geometric crossover operators combine the behaviors of parents on particular tests. For instance, in the exact geometric crossover in the Boolean domain (Fig. 5.3a and (5.8)), a mixing random subprogram decides, for each test individually, which parent to copy the output from. According to the convention adopted in this book, such outputs are column vectors (see, e.g., Figs. 7.5), and a mixing program, by picking elements from such vectors, effectively splice them into smaller vertical segments – hence the name.

Execution records facilitate also *horizontal*, column-wise problem decomposition, i.e. along the course of program execution. In the median example in Fig. 7.2, horizontal decomposition consists of splitting the original task into two separate subtasks of (i) sorting the list and (ii) retrieving the central element of the sorted list (or averaging the pair of the central elements for the even-length lists). We argue that such desirable decompositions can be automatically derived by analyzing execution records for entire *populations* of programs.

To proceed with our argument, we need to consider the joint behavioral space of multiple programs. Figure 11.1 visualizes the behavior of three programs that start with the same input and end up with the same output. Contrary to previous figures, where a graph node corresponded one-to-one to a state of an execution environment, here it represents a *combination* of execution states for all tests in $T$ (a $|T|$-ary Cartesian product of execution states). We term such a combination a *c-state*. For instance, $s_1$ is the combination of executions states reached by program $p_1$ after executing its first instruction for all available tests. Consistently, a path in the graph represents a *combined trace* ($c$-trace) and captures the behavior of a program for all tests. $c$-traces