AVMf: An Open-Source Framework and Implementation of the Alternating Variable Method

Authors and institutions
suppressed for double-blind review

Abstract. The *Alternating Variable Method* (AVM) has been shown to be a particularly fast and effective local search technique for search-based software engineering. Recent improvements to the AVM have generalized the types of representations it can optimize and have provably improved its runtime for certain types of objective function landscape. Until now, however, there has been no publicly-available code implementation of these algorithms. This paper introduces AVMf, an object-oriented Java framework that provides such an implementation. AVMf is ready for download and configuration for use in a wide variety of SBSE projects.

1 Introduction

The *Alternating Variable Method* (AVM) is a local search method that was first proposed for an SBSE problem by Bogdan Korel in 1990 [9] for generating numerical software test data. Despite the application of supposedly more robust, global search techniques to this problem (e.g., Genetic Algorithms), the AVM has stood the test of time. In 2007, Harman and McMinn [5] reported its effectiveness and efficiency for a series of C programs, and combined it with a GA to provide a "best of" Memetic Algorithm approach [6]. It has since been used to generate test suites for Java programs [3], having been implemented into the EvoSuite tool; generate rows of data for testing databases [7,12] with the *SchemaAnalyst* tool; and combined with dynamic symbolic execution in Microsoft's Pex tool [11]. The AVM has since found application to problems outside of test data generation, for example decision ordering for engineering software product lines [16], balancing workload in requirements assignment [15], reliability-redundancy-allocation problems [14], as well as test case selection [13] and prioritization [2].

Since the publication of Korel's original paper in 1990, the AVM algorithm has been extended and improved for problems in SBSE: it can now handle a wider range of variable types, including fixed-point numbers [5] and strings [12], and can it incorporate new strategies that are proven to speed up the search for certain common types of objective function landscape [8].

The AVM is therefore capable of handling a variety of search representations and locating solutions to SBSE problems in a very efficient manner. Nevertheless, to incorporate it into an SBSE project, an SBSE developer has previously had to understand the different variants of the algorithm and then produce a faithful implementation of it — a potentially time-consuming and error prone task. To address this problem, we have developed AVMf, an open-source object-oriented framework that implements variants of the AVM and its representations, and is available for download and deployment in SBSE projects. AVMf is fully documented and comes with a series of examples demonstrating its usage.

2 The AVM and Recent Improvements to the Algorithm

The original AVM. The AVM optimizes a vector $\vec{x} = (x_1, \ldots, x_{len})$ according to some objective function by taking each variable $\vec{x}_i, 0 \leq i \leq len$ of the vector in turn and subjecting it to an individual search process. The original AVM used a variable search process subsequently named "Iterated Pattern Search" (IPS) [8], shown by lines 1–7 of Figure 1. The initial part of this algorithm involves making an increase and decrease of 1 to the value of the variable (lines 2–3), referred to as *exploratory moves*. If either exploratory move leads to an improvement in the objective value for the complete vector, a positive or negative "direction" is established for making further *pattern moves* (lines 4–6). Pattern moves of increasing size continue to be made while the objective value improves. When a pattern move does not improve on the objective value compared to the previous move, the search has likely overshot the optimum, due to a pattern move that was larger than the difference between the current value of $\vec{x}_i$ and the optimal value. When this occurs, IPS loops back to the exploratory move process to re-establish a new direction. If exploratory moves do not lead to an improvement in objective value, IPS terminates and hands back control to the main loop, thus leading to the consideration of the next variable in the vector.

When all variables in the vector have been considered, the AVM wraps back to the first. When a cycle of all variables has completed without any improvement in the objective function, the AVM is lodged in a local optimum. At this point the search process can be restarted with a new (typically random) series of vector values. The AVM continues in this fashion until resources are exhausted (e.g., a maximum number of objective function evaluations or restarts have been expended, or a time limit has expired), or, the best outcome is attained — the optimal target vector is discovered. (For simplicity, these different termination criteria are not included as part of the algorithm definitions in Figure 1.)

New Representations. Korel only demonstrated the original AVM with integer variables [9]. Harman and McMinn [5] extended this initial definition by allowing each variable to be specified with a set number of decimal places p, allowing fixed-point numbers to be handled. Exploratory moves correspond to the smallest possible increments and decrements of the variable (i.e., $\pm 10^{-p}$). McMinn et al. [12] allowed string variables to be represented by the approach. A string variable is essentially a sub-vector, whose elements are characters that are individually manipulated by the local search routine. The length of this sub-vector is allowed to vary through a special sequence of moves that increase and decrease its size, allowing for the optimization of variable-length strings.

New Variable Search Algorithms. Kempka et al. [8] proposed two new variable searches for the AVM, as shown in Figure 1. Kempka et al. proved that these search techniques are more efficient than IPS for unimodal objective function landscapes. "Geometric Search" (GS) begins by performing exploratory moves followed by pattern moves like IPS. Unlike IPS, however, it does not iterate after overshooting the optimum. Instead it uses past moves to "bracket" the upper and lower limits of the variable in which the optimum must lie, performing a binary search to finally locate it (lines 8–15 of Figure 1). "Lattice Search" (LS) is

```
1:  while true do                                                          ▷ {IPS}
2:      if obj(x − 1) ≥ obj(x) and obj(x + 1) ≥ obj(x) return x            ▷ {IPS, GS, LS}
3:      if obj(x − 1) < obj(x + 1) then let k := −1 else let k := 1        ▷ {IPS, GS, LS}
4:      while obj(x + k) < obj(x) do                                       ▷ {IPS, GS, LS}
5:          let x := x + k, k := 2k                                        ▷ {IPS, GS, LS}
6:      end while                                                          ▷ {IPS, GS, LS}
7:  end while                                                              ▷ {IPS}
8:  let ℓ := min(x − k/2, x + k), r := max(x − k/2, x + k)                  ▷ {GS, LS}
9:  while ℓ < r do                                                         ▷ {GS}
10:     if obj(⌊(ℓ + r)/2⌋) < obj(⌊(ℓ + r)/2⌋ + 1) then                    ▷ {GS}
11:         r := ⌊(ℓ + r)/2⌋                                               ▷ {GS}
12:     else                                                               ▷ {GS}
13:         ℓ := ⌊(ℓ + r)/2⌋ + 1                                           ▷ {GS}
14:     end if                                                             ▷ {GS}
15: end while                                                              ▷ {GS}
16: let n := min{n | F_n ≥ r − l + 2}                                      ▷ {LS}
17: while n > 3 do                                                         ▷ {LS}
18:     if ℓ + F_{n−1} − 1 ≤ r and obj(ℓ + F_{n−2} − 1) ≥ obj(ℓ + F_{n−1} − 1) then   ▷ {LS}
19:         let ℓ := ℓ + F_{n−2}                                           ▷ {LS}
20:     end if                                                             ▷ {LS}
21:     let n := n − 1                                                     ▷ {LS}
22: end while                                                              ▷ {LS}
23: x := ℓ                                                                 ▷ {GS, LS}
```

Fig. 1. IPS, LS, GS search algorithms for the AVM for a variable $x \in D$. $obj(x)$ is the objective function, computed for the complete vector $\vec{x}$ currently being optimized by the AVM, where $\vec{x}_i := x, 0 \leq i \leq len$. F is the Fibonacci sequence starting from $F_0 = 0$. Each line is annotated to show the algorithm(s) to which it belongs.

a slightly faster alternative to GS (provided the objection function landscape is, again, unimodal). LS is similar to GS, except that after bracketing the optimum, it converges upon it through moves that increase the value of $\vec{x}$ from the lower end by adding Fibonacci numbers (lines 16–22 of the figure).

3 The AVM Framework (AVM*f*)

The AVM Framework (AVM*f*) implements both the AVM algorithm and the subsequent enhancements to the original version proposed by Korel. The framework has been implemented with the aim of making the core algorithms as clear as possible, thereby closely matching the algorithmic definitions of Figure 1, while still adhering to well-accepted principles of good object-oriented design. AVM*f* is publicly available at `https://github.com/AVMf/avmf` as a Git repository for inclusion in SBSE projects where the AVM may be the core search algorithm, or, a component of a more complex technique (e.g., a Memetic Algorithm) involving calls to algorithms in the framework. Or, the code can simply be lifted from the repository and adapted to a project as developers see fit.

To enable its algorithms to be easily used in SBSE projects, AVM*f* provides a framework of Java classes, which we now describe in detail. Each aspect of the framework is practically demonstrated by the source code of a series of examples in the repository, which are introduced at the end of this section.

Configuring an AVM search. The primary class is the `AVM` class in the root (i.e., `org.avmframework`) package. In order to construct an `AVM` instance, the developer must supply an instance of one of the variable search methods;

IteratedPatternSearch, GeometricSearch or LatticeSearch, which reside in the localsearch package. The developer must also construct the AVM instance using a TerminationPolicy parameter, an object that decides when the AVM should terminate if a solution cannot found. Options include a maximum number of objective function evaluations, a maximum number of restarts, or a time limit. Finally, constructing the AVM instance further requires two objects of type Initializer that are used to initialize variable vector values at the start of the search and re-initialize them on a restart. Default values may be used that can be specified for each variable, or random values can be chosen (through instances of either DefaultInitializer or RandomInitializer, two classes that both reside in the initializer package). To support the generation of random numbers, AVM*f* requires a RandomGenerator from the org.apache.commons library that provides an implementation of the Mersenne Twister algorithm.

In order to initiate a search process, the search method of the AVM instance must be invoked with an instance of a Vector class and an ObjectiveFunction, respectively. The Vector class describes the representation of the problem (i.e., the types of variables in the vector to be optimized), while the ObjectiveFunction class describes how instances of those vectors should be rewarded with objective values during the search.

Representation. In order to configure the search representation, an instance of the Vector class (in the root package) must be created, and variables added to it through the addVariable method, which accepts an instance of a Variable. Since the Variable class is abstract, an instance of one of its concrete subclasses must be provided (i.e., one of IntegerVariable, FixedPointVariable, CharacterVariable or StringVariable). Each variable must be constructed with information such as its minimum or maximum value (maximum length for strings), number of decimal places for fixed-point variables, and a "default" initial value in the search space (e.g., an empty string or a zero value). These values are used to initialize vector variables when the DefaultInitializer is used to providing a starting point for the search, as previously described in this section.

Objective Function. In contrast to the rest of the framework, which requires configuring instances of existing classes, an objective function must be supplied to the search process by overriding the abstract ObjectiveFunction of the objective package. This involves providing an implementation of the computeObjectiveValue method that takes a Vector as a parameter and returns an instance of the abstract ObjectiveValue class. Since the AVM only needs to know whether one entity has a "better" objective value than another, exact numerical values are not needed, and so this class requires the "betterThan", "worseThan" and "sameAs" methods to be overridden. The objective package also supplies the concrete NumericalObjectiveValue class for returning higher-is-better or lower-is-better numerical objective values as needed.

Reporting. The search method of the AVM class returns an instance of the Monitor class, which can be used to find out interesting statistics regarding the search. These include the best vector found by the search, its objective value, the number of objective function evaluations that took place, the number

of restarts that took place, and the amount of time that the search took (in milliseconds). The `Monitor` class can also report the number of *unique* objective function evaluations. The objective function can make optional usage of a cache that maps previously "seen" vectors to objective values, avoiding the need to perform potentially costly re-evaluations.

Examples. AVM*f* comes with three accessible examples demonstrating its use. (Instructions on how to compile and run these examples are available in the project's `README.md` file located in the main directory of the project's repository.) The first, `Quadratic`, demonstrates the use of the AVM to solve a quadratic equation by finding one of its roots. `AllZeros` demonstrates the optimization of an array of integers to zero values from arbitrary random values, while `String` optimizes a string value from an initially random string to a specified target.

Each example makes use of its own problem-specific fitness function that can be seen as part of its code definition. Although not directly connected to SBSE, the following self-contained example is taken from the `Quadratic` class, where the constants `A`, `B` and `C` correspond to the co-efficients of the equation (in the example `A = 4`, `B = 10` and `C = 6`). The function obtains the value of `x` from the (single variable) vector, and computes the value of `y`. The objective value is then assigned as the distance between `y` and zero, since intuitively, the closer the value of `y` to zero, the closer the search is to finding one of the roots of the equation.

```
ObjectiveFunction objFun = new ObjectiveFunction() {
    @Override
    protected ObjectiveValue computeObjectiveValue(Vector vector) {
        double x = ((FloatingPointVariable) vector.getVariable(0)).asDouble();
        double y = (A * x * x) + (B * x) + C;
        double distance = Math.abs(y);
        return NumericObjectiveValue.LowerIsBetterObjectiveValue(distance, 0);
    }
};
```

The following is the output of the search process, showing how the AVM search correctly found one of the roots (i.e., -1.5). Re-running the search from different starting positions leads to the other root, -1, also being found.

```
Best solution: -1.5
Best objective value: 0.0
Number of objective function evaluations: 80 (unique: 80)
Running time: 3ms
```

4 Conclusions and Future Work

This paper introduced AVM*f*, an open-source implementation of the AVM and a framework supporting its use in SBSE projects. AVM*f* offers features associated with cutting-edge local search techniques, such as a representation for a wide variety of data types and the integration of new variable search algorithms. Since a well-documented version of AVM*f* is now freely available for download at `https://github.com/AVMf/avmf`, we judge that it can advance the use of the AVM in both industrial practice and in the SBSE research community. Moreover, AVM*f*'s extensible object-oriented design will enable others to implement search-based solutions to many different problems in the field of software engineering. Using AVM*f*, possible future applications of the AVM include the following:

Automatically Generating Readable Test Data. Generating readable tests that humans can easily understand has been a recent interest of search-based testing researchers (e.g., rewarding inputs that obtain a high score from a language model [1]). In a recent study evaluating test generation tools, participants also requested more readable values [4]. Given that the AVM employs a local search, it could start with examples of human-generated inputs and adapt them to new coverage targets — all without losing the qualities of the original data.

Automatically Determining Optimal Software Configuration Values. Highly configurable software tools, such as the GCC compiler, may be tunable through the use of search-based techniques such as genetic algorithms or the AVM [10]. In large search spaces of parameters, the AVM's exploratory move phase equips it to quickly discover which particular variables are relevant to the problem, while its phase of pattern moves allows it to determine the optimal values of parameters. Again, as a local search technique, the AVM is also well suited to taking an existing known-good human solution and improving upon it.

Automated Bug-Fixing. Recent experiments reveal that real-world bugs can occur as a result of mistakes made when defining constant variables and setting values in configuration files [17]. As such, the AVM could search for appropriate values that could potentially form the basis of a "fix". During its exploratory move phase the AVM could, by performing a quick sweep of small changes through the values involved and seeing how the resulting fitness values are affected, quickly determine which constants are relevant to the fix.

References

1. Afshan, S., McMinn, P., Stevenson, M.: Evolving readable string test inputs using a natural language model to reduce human oracle cost. In: ICST (2013)
2. Arrieta, A., Wang, S., Sagardui, G., Etxeberria, L.: Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In: GECCO (2016)
3. Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. JSS (2015)
4. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: ISSTA (2013)
5. Harman, M., McMinn, P.: A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: ISSTA (2007)
6. Harman, M., McMinn, P.: A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEE TSE (2010)
7. Kapfhammer, G.M., McMinn, P., Wright, C.J.: Search-based testing of relational schema integrity constraints across multiple database management systems. In: ICST (2013)
8. Kempka, J., McMinn, P., Sudholt, D.: Design and analysis of different alternating variable searches for search-based software testing. TCS (2015)
9. Korel, B.: Automated software test data generation. IEEE TSE (1990)
10. Kukunas, J., Cupper, R.D., Kapfhammer, G.M.: A genetic algorithm to improve Linux kernel performance on resource-constrained devices. In: GECCO (2010)
11. Lakhotia, K., Tillmann, N., Harman, M., Halleux, J.: FloPSy — search-based floating point constraint solving for symbolic execution. In: ICTSS (2010)
12. McMinn, P., Wright, C.J., Kapfhammer, G.M.: The effectiveness of test coverage criteria for relational database schema integrity constraints. ACM TOSEM (2015)
13. Pradhan, D., Wang, S., Ali, S., Yue, T.: Search-based cost-effective test case selection for manual execution within time budget: An empirical study. In: GECCO (2016)
14. Qiu, X., Ali, S., Yue, T., Zhang, L.: Reliability-redundancy-location allocation with maximum reliability and minimum cost using search techniques. IST (2016)
15. Yue, T., Ali., S.: Applying search algorithms for optimizing stakeholders familiarity and balancing workload in requirements assignment. In: GECCO (2014)
16. Yue, T., Ali, S., Lu, H., Nie., K.: Search-based decision ordering to facilitate product line engineering of cyber-physical system. In: MODELSWARD (2016)
17. Zhong, H., Su, Z.: An empirical study on real bug fixes. In: ICSE (2015)