Theoretical Computer Science 605 (2015) 1–20

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Design and analysis of different alternating variable searches for search-based software testing[☆], [☆☆]

Joseph Kempka, Phil McMinn[1], Dirk Sudholt[*,2]

Department of Computer Science, University of Sheffield, Sheffield, S1 4DP, UK

ARTICLE INFO

Article history:
Received 26 November 2013
Received in revised form 17 November 2014
Accepted 11 December 2014
Available online 8 January 2015

Keywords:
Search-based software engineering
Test data generation
Local search
Runtime analysis
Theory

ABSTRACT

Manual software testing is a notoriously expensive part of the software development process, and its automation is of high concern. One aspect of the testing process is the automatic generation of test inputs. This paper studies the Alternating Variable Method (AVM) approach to search-based test input generation. The AVM has been shown to be an effective and efficient means of generating branch-covering inputs for procedural programs. However, there has been little work that has sought to analyse the technique and further improve its performance. This paper proposes two different local searches that may be used in conjunction with the AVM, Geometric and Lattice Search. A theoretical runtime analysis proves that under certain conditions, the use of these searches results in better performance compared to the original AVM. These theoretical results are confirmed by an empirical study with five programs, which shows that increases of speed of over 50% are possible in practice.

1. Introduction

Testing software for functional correctness can be classified as two distinct types of activity, *black-box testing*—where the software is exercised with respect to a specification, formal or otherwise—and *white-box testing*, where test inputs are derived with respect to the software's underlying program code. Black-box and white-box testing are complementary. While black-box testing can check that all expected functionality is present, i.e., trapping so-called *faults of omission*, white-box testing can check for *faults of commission*—unintended software behaviours not documented in the specification, but present in the implementation.

The degree to which white-box testing has been performed is measured by coverage metrics. The strongest coverage metric is *path coverage*. A program "path" is simply the sequence of program statements executed through a program, decided by the outcomes of decisions at conditionals statements such as `if`, `while` and `for` etc., and path coverage is the proportion of all possible paths through the software that were exercised by the tests. However, for any realistic program, full path coverage is an unrealistic target due to the potentially infinite number of paths that may be involved [1]. A more achievable and common goal is *branch coverage*, where every decision in the program is executed as true and false.

[☆] This article was originally scheduled to be compiled on the Special issue Evolutionary Computation 2013, now getting compiled in to a regular issue.
[☆☆] A preliminary conference version with parts of the results appeared at GECCO 2013 [11].
[*] Corresponding author.
[1] This work is funded in part by the EPSRC project "RE-COST" (grant no. EP/I010386).
[2] The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007–2013) under grant agreement no. 618091 (SAGE).

http://dx.doi.org/10.1016/j.tcs.2014.12.009
0304-3975/© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Software testing is an expensive and labour-intensive process. One potential way to lower the costs involved in testing is to generate inputs for achieving white-box coverage automatically, and a technique that has been of particular interest to researchers in recent years is that of *search-based test input generation* [17,13,16,26]. In search-based testing, stochastic optimisation techniques are employed to drive the dynamic executions of a piece of software towards the coverage of a certain test goal, for example the execution of a particular program branch. Due to the non-linearity of software, it is usually non-obvious which inputs will execute which branches. Search-based testing reformulates the predicates in a program that guard the execution of a branch into a fitness function, such that inputs closer to satisfying those predicates are awarded better fitness values. These fitness functions may then be optimised by an optimisation technique, such as a randomised search heuristic, to locate an input for a particular branch.

A simple local search optimisation technique that has been shown to be effective for covering branches of procedural programs is the *Alternating Variable Method* (AVM) [13]. In a recent empirical study by Harman and McMinn with a range of C programs, the AVM was able to cover the majority of branches faster than a genetic algorithm [9]. This suggests that the underlying fitness landscape for covering individual program branches is relatively simple most of the time, with more "heavyweight" population-based approaches like Genetic Algorithms only required in a minority of cases [19]. Despite this, there has been relatively little work devoted to analysing and improving the performance of the AVM technique.

The AVM can be regarded as a general framework in which a local search strategy is applied in turn to each individual input vector variable of the program under test. In this paper, we view the local search strategy to be a component of the overall framework that may be substituted for another. The original AVM applied an accelerated hill climb that we refer to as *Iterated Pattern Search* (IPS), where "exploratory" moves in a direction of fitness improvement are proceeded by "pattern" steps in the same direction. Pattern steps iteratively increase in size for as long as improvements in fitness continue. As soon as fitness peaks, exploratory moves are again made to re-establish a new direction.

In this paper, we propose to replace IPS with two different approaches for exploring individual dimensions of the input vector—*Geometric Search* and *Lattice Search* (see Section 5 for details and formal definitions). Geometric and Lattice Search are elimination searches that are able to find the optimum of a one dimensional function that is unimodal on a given interval. They work by comparing the fitness values of two points at predetermined positions, using the result to select a new but smaller sub-range. The algorithm iterates until it is left with one point. Geometric Search splits the range in two by comparing two positions in the middle of the interval to determine whether an optimum is contained in the bottom or top half of the interval. Lattice Search compares points that are offset by Fibonacci numbers.

We examine all three variants of the AVM through theoretical runtime analyses and through empirical experiments, in order to understand and improve the performance of this popular approach to search-based test input generation.

While prior theoretical runtime analyses of the original AVM with IPS involved specific programs and branches [2,3], we furnish a more general result, proving that for all unimodal functions Geometric and Lattice Search are faster than IPS, when used in the framework of AVM. In a more general sense, Geometric and Lattice Search converge faster to local optima than IPS. These theoretical results are complemented by an empirical study on open source programs, involving unimodal as well as non-unimodal fitness landscapes. On most branches the alternative local searches perform significantly better than IPS. This includes unimodal landscapes, in agreement with our theory, as well as non-unimodal ones, where the assumptions of our theory are not met. This indicates that faster convergence to local optima is beneficial on a broad range of instances. The only departure from this pattern was found for one complex landscape of a type for which—as observed in prior studies [9,19]—a Genetic Algorithm is significantly better than AVM. A further theoretical analysis for this challenging branch shows that worse performance is due to the kind of local optima being returned by local search. On this branch, the alternative local searches are faster in finding *some* local optimum, but *which* local optimum is returned has an adverse effect on global search performance in this specific case.

The contributions of this paper are therefore as follows:

1. The incorporation of two different local searches *Geometric Search* and *Lattice Search* into the AVM approach for finding test inputs for programs (Section 5).
2. A theoretical runtime analysis of the local search used in the original AVM approach, IPS (Section 4), and Geometric and Lattice Search, on unimodal functions. Starting at distances up to d from the global optimum, IPS has a worst-case (regarding the choice of the starting point) running time of $\Theta((\log d)^2)$ on every unimodal function. The same holds for its average-case performance on a simple unimodal function. Geometric and Lattice Search perform well on all unimodal functions as they only need time $O(\log d)$ in the worst case (Sections 5.1 and 5.2).
3. An empirical analysis of the AVM, comparing IPS, Geometric and Lattice Search on five programs, including unimodal and non-unimodal functions, complementing our theoretical results on unimodal functions and providing additional evidence that the alternative local searches also speed up search on non-unimodal functions, possibly due to their faster convergence to local optima (Section 6).
4. A theoretical analysis and discussion for landscapes with multiple local optima highlighting how global search performance can be affected by the question *which* local optimum is being returned by local search (Section 7).

```
        int compare(int x, int y) {
(1)       if (x < y)  {
(2)         return -1;
          }
(3)       if (x == y) {
(4)         return 0;
          }
(5)       return 1;
        }
```

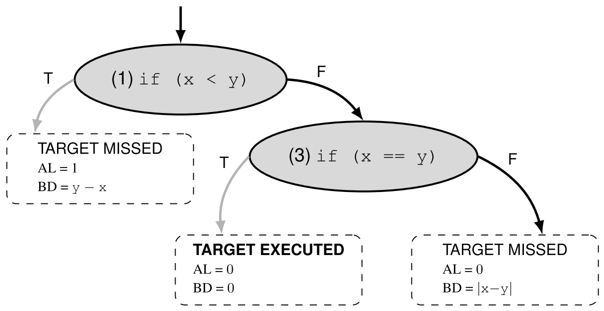


Fig. 1. Example program for which branch-covering test inputs are required, and a partial control flow graph (CFG) showing fitness function computation (AL = approach level, BD = branch distance) for execution of the true branch from the second "`if`" statement of the program at CFG node (3) (corresponding control flow graph node numbers appear to the left of relevant code statements).

2. Background

We introduce the fitness function and the representation used in this paper for search-based test input generation, the AVM algorithm, and important background to theoretical runtime analysis.

2.1. Representation and fitness function

The fitness function for covering individual program branches is a multivariate function $mf(\vec{x}) \rightarrow \mathbb{R}$, that takes an input vector $\vec{x} = (x_1, x_2, \ldots, x_n)$, i.e., an ordered list of arguments that are passed to a procedure. In this paper, we assume $\vec{x}$ can be modelled as a sequence of integers. Our contributions naturally extend to rational numbers, when these are represented using fixed-point representations (i.e., a fixed number of digits is used for both integer and fractional parts of a number).

The fitness function measures how "close" an input vector was to executing a target branch. It is minimised by the search, with a zero value indicating an input that covers the branch. The fitness function has two components, which we describe with the aid of the example in Fig. 1 and the execution of the true branch from control flow graph node (3).

The *approach level* relates to the decision points in the program appearing en route to reaching the target branch, and reflects the number of control flow graph (CFG) nodes unexecuted by $\vec{x}$ on which the target branch is directly or transitively control dependent. These are decision nodes in the CFG that must be executed with a specific outcome, otherwise the target cannot be reached. For example, in Fig. 1 the target branch is control dependent on the `if` statements at CFG nodes (1) and (3), as demonstrated by the (partially-drawn) CFG. If the "wrong" choices are made at either of these decision points (i.e., the condition evaluates to true for node (1) or false for node (3)), the target is unexecuted by the path taken through the control structure of the program. If the true branch is taken at CFG node (1), the approach level is 1, since node (3) is unexecuted. If the false branch is taken from node (1), the approach level is 0, since all control dependencies will be executed.

The *branch distance* is computed from the values of variables at predicates where control flow diverged from the target at some control dependent CFG node. It is intended as a measure of "how far" the input was from executing a condition in the required way, so that control flow is directed towards, rather than away from the current target in question. For example, in Fig. 1, if the input takes the true branch at CFG node (1), the branch distance is computed using the formula $\mathtt{y} - \mathtt{x}$. With this formula, inputs are rewarded on the basis of how close they are to making `y` equal `x`, which would result in the alternative (and desired) false branch being taken instead of the true branch. If the search succeeds in finding an input that results in the false branch being taken at CFG node (1), but then takes the false branch at CFG node (3), the branch distance is computed using the formula $|\mathtt{x} - \mathtt{y}|$. This formula rewards inputs that are close to being equal, in order to encourage the search towards inputs that result in the desired true branch to be taken in the execution path.

The description of branch distance computation presented in the previous paragraph and in Fig. 1 is a simplified view designed to convey its intuition. A full list of rules and formulas for computing branch distance with different predicate types, originally due to Tracey [29], is presented in Table 1. In essence, the branch distance should always be a positive value when the branching condition is *not* executed as desired, else zero. Therefore, the addition of a non-zero constant K ($K = 1$ for experiments in this paper) is required in order to accommodate for special cases where a "raw" branch distance calculation would return zero even when the predicate is still false. For example, for executing the predicate $\mathtt{x} < \mathtt{y}$ as *true* (as opposed to false in the example of Fig. 1), the formula $\mathtt{x} - \mathtt{y}$ returns 0 when $\mathtt{x} = \mathtt{y}$ (that is, when $\mathtt{x} < \mathtt{y}$ is *not* true). Thus using $r + K$, where r is the raw branch distance value (i.e., the value of $\mathtt{x} - \mathtt{y}$ for $\mathtt{x} < \mathtt{y}$ as true), always ensures a distance greater than zero when the condition is not executed as desired. The addition of K is included in all branch distance formulas, whether it is strictly needed to distinguish special cases such as these or not. In the table, a maps to the left hand side of the predicate, and b the right. For instance, for the branching predicate `x % y == 3` ("`x % y`" meaning "x mod y"), a becomes `x % y` and b becomes `3`. Due to the possibility of short-circuiting (e.g. with the C operators "`||`" and "`&&`"), predicates consisting of disjuncts and conjuncts require special handling. For brevity, we refer interested readers to reference [18].

Table 1
Tracey's branch distance functions for relational predicates (from reference [29]). The value K, $K > 0$, refers to a constant which is always added if the predicate is not true.

Relational predicate	Branch distance computation
Boolean	if TRUE then 0 else K
$a = b$	if $\lvert a - b\rvert = 0$ then 0 else $\lvert a - b\rvert + K$
$a \neq b$	if $\lvert a - b\rvert \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$

In practice, the maximum branch distance varies from branch to branch, depending on the types of the variables involved in the branch predicate. Instead of applying costly analysis to determine variable types, it is easier, in practice, to simply apply a generic normalisation function. In this paper we use the function $\mathrm{norm}(d(\vec{x})) = 1 - 1.001^{-d(\vec{x})}$, where $d(\vec{x})$ returns the value of $r + K$ for $\vec{x}$ and $\mathrm{norm}(d(\vec{x}))$ is the normalised branch distance. The complete fitness value is computed by normalising the branch distance and adding it to the approach level, i.e., $mf(\vec{x}) = \mathrm{AL}(\vec{x}) + \mathrm{norm}(d(\vec{x}))$ [31]. As the co-domain of norm is $[0, 1)$ and the approach level takes integer values, the latter always represents the dominating term in the fitness function. In other words, a search point $\vec{x}$ has a better fitness than $\vec{y}$ if its approach level is smaller, or if they have equal approach levels, but $\vec{x}$ has a smaller branching distance. The choice of the normalisation function is not important when only the relative ranking of solutions is of concern—as in this paper—and identical rankings are produced (e.g., when the constant 1.001 is replaced in norm with any other base greater than 1); or, the lexicographic ordering of a bi-objective function consisting of the approach level and the branch distance is minimised.

2.2. The alternating variable method (AVM)

The AVM can be viewed as a general framework that proceeds from a random starting point in the search space, and works by calling a local search function on each element of the input vector in turn. That is, while the local search is performing "moves" on one component of the vector, the values for all other dimensions remain fixed. If, during this search, a fitness of zero is found, the AVM terminates with the branch-covering input. If, however, a local optimum is reached, the AVM advances to the next element. If the fitness cannot be improved after cycling through all elements, the AVM restarts from another randomly-generated input, continuing the search for a branch-covering input until the number of fitness function evaluations exceeds a predefined maximum.

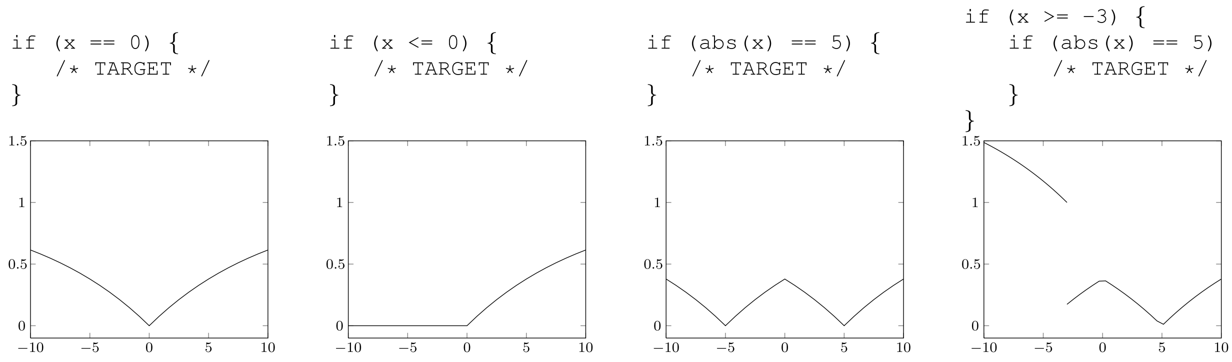
Algorithm 1 describes the AVM framework more formally. The initial vector is chosen uniformly at random using the function **random**(). The algorithm transforms the multivariate fitness function mf into a one dimensional projection f (line 5). The function f is equivalent to evaluating mf with an input vector where all components except x_i are set to constants and x_i is substituted by the free parameter x. The function f is passed to a local search algorithm called **local_search**, along with x_i, the starting point for the search. The variable c counts how many variables AVM has optimised since the last improvement of mf; it restarts with a uniform random value for $\vec{x}$ once it has cycled through all n variables with no successful improvement in fitness. The fitness function keeps track of the number of fitness evaluations, maintaining a mapping of previously evaluated vectors to their corresponding fitness values. Once a branching-covering input vector is found, or when the number of evaluations exceeds the maximum (i.e., the search has failed), an exception is raised to terminate the search (not shown in our algorithms for space and simplicity).

Algorithm 1 The AVM framework for optimizing $\vec{x} = (x_1, \dots, x_n)$.

```
1:  while true do
2:      let $\vec{x} := $ random()
3:      let $i := 1$, $c := 0$
4:      while $c < n$ do
5:          let $f : x \mapsto mf(x_1, \dots, x_{i-1}, x, x_{i+1}, \dots, x_n)$
6:          let $\vec{x}' := $ local_search($f, x_i$)
7:          if $mf(\vec{x}') < mf(\vec{x})$ then
8:              let $\vec{x} := \vec{x}'$, $c := 0$
9:          else
10:             let $c := c + 1$
11:         let $i := (i \bmod n) + 1$
```

2.3. Runtime analysis and related work

The aim of runtime analysis is to provide mathematical estimations of the running time of algorithms such as randomised search heuristics, in order to provide a better understanding of their underlying working principles and to help design better



```
if (x == 0) {
    /* TARGET */
}
```

```
if (x <= 0) {
    /* TARGET */
}
```

```
if (abs(x) == 5) {
    /* TARGET */
}
```

```
if (x >= -3) {
    if (abs(x) == 5) {
        /* TARGET */
    }
}
```

(a) strictly unimodal (b) strictly unimodal with additional global optima (c) weakly unimodal (d) multimodal

Fig. 2. Examples of different fitness landscapes, including simple example code for a single variable x that generates these landscapes; `abs(x)` denotes the absolute value of x. To ease presentation, we used $K = 0$ in the branch distance and the base 1.001 of the normalisation function was changed to 1.1 for generating these plots. The function in (b) results from the strictly unimodal function (a) and adding global optima for $x < 0$. In (d) values $x < 3$ lead to a high fitness value because of the worse approach level.

algorithms. Runtime analysis has established itself as a leading theory in randomised search heuristics, with many new results in the last 15 years [25,4,10]. In particular, in the area of search-based software engineering recent results include computing input-output sequences [14,15], test input generation [3], and software project scheduling [21,22]. The following paragraphs review related work in runtime analysis.

Arcuri, Lehre, and Yao [3] were the first to present a runtime analysis of search-based input generation. They focussed on the triangle classification problem, which involves three integer variables describing side lengths of a potential triangle. The task is to classify the input as a scalene, equilateral, or isosceles triangle, or not representing a triangle. Their analysis was limited to the time for covering the *equilateral* branch of the problem. If the range of input values contains n numbers, the expected time of random search is $\Theta(n^2)$. Hill climbing needs expected time $\Theta(n)$, i.e., in every iteration a single variable is increased or decreased by 1. For the AVM they proved an upper bound of $O((\log n)^2)$ and a weaker lower bound of $\Omega(\log n)$ [3].

Later on, Arcuri [2] extended the analysis of the AVM to all branches of the triangle classification problem, showing that the expected running time is bounded from above by $O((\log n)^2)$ on all branches. Certain branches with many global optima (cf. Fig. 2(b) on p. 5) only need $O(\log n)$ time. In this paper we extend his results by proving an upper bound of $O((\log n)^2)$ for *all* strictly unimodal functions (and functions with further global optima).

The performance of search heuristics on unimodal functions was also studied by Dietzfelbinger, Rowe, Wegener, and Woelfel [6,5]. They showed that algorithms that randomly displace the current position of a search algorithm using any fixed probability distribution need at least $\Omega((\log d)^2)$ time in the worst case, when the initial distance to the optimum is chosen from $\{1, \ldots, d\}$. Thus, no algorithm using such a fixed distribution can do better than the original AVM. In other words, every local search strategy aiming at better performance needs to use a probability distribution, or search strategy, that depends on the initial point. This is the case for all AVM variants considered here.

3. Preliminaries

Our theoretical analyses focus on the time for finding a local optimum using local search within the framework of the AVM. Recall that these local searches are executed on one coordinate of the multivariate fitness function. We count the number of function evaluations, also called *fitness evaluations*, made by local search until an optimum is found for the first time. The motivation for considering fitness evaluations is that such an evaluation is the most costly operation as it involves executing the program. In some cases we count *unique fitness evaluations*, i.e., the number of *different* search points evaluated. This reflects the fact that it is easy to cache past evaluations, so evaluating the same point twice only incurs insignificant additional cost.

In our analyses we consider local search minimising a function $f : D \to \mathbb{R}$ for a finite domain $D \subset \mathbb{Z}$, which can be regarded a projection of the multivariate fitness function mf. For ease of presentation, we assume $f(x) = \infty$ for all $x \notin D$. This includes settings where f is the branch distance before or after normalisation. As already noted in Section 2.1, the precise choice of a normalisation function is irrelevant in our work; all local searches analysed in this work only use information about the ranks of search points. So every strictly increasing normalisation function leads to the same sequences of search points queried and hence the same performance.

In previous work [2,3] the authors derived performance results with regard to the size n of the domain, e.g., $\{1, \ldots, n\}$ or $\{-n/2 + 1, \ldots, n/2\}$. Here we consider the initial distance d to the optimum instead, as this distance governs the running time of all local searches considered in this work. As $d \le n$ upper running time bounds using d are generally stronger and more precise than those using n.

In the remainder, we will deal with the following function classes, illustrated in Fig. 2. A function $f: D \to \mathbb{R}$ with domain $D \subseteq \mathbb{R}$ is called *strictly unimodal* if it only has a single local optimum. If f is to be minimised (the case of maximisation is symmetric) this means that

$$f(\ell_1) > f(\ell_2) > f(\text{opt}) < f(r_1) < f(r_2) \tag{1}$$

for all $\ell_1 < \ell_2 < \text{opt} < r_1 < r_2$, where $\text{opt} = \min(f)$ is the global minimum of f. In other words, f is strictly decreasing on $D \cap (-\infty, \text{opt}]$ and strictly increasing on $D \cap [\text{opt}, \infty)$. The upper bounds obtained in the remainder for the running time of AVM on strictly unimodal functions also apply to functions f that result from taking a strictly unimodal function f' and assigning function values $\min(f')$ to further points, formally:

$$\forall x \in D: f(x) \in \left\{f'(x), \min\left(f'\right)\right\}. \tag{2}$$

Fig. 2(b) shows one such example. Note that in general global optima are not required to form one interval.

In order to classify real-world fitness landscapes in Section 6, we also consider *weakly unimodal* functions, defined as functions where all local optima are also global optima. In contrast to strictly unimodal functions with additional optima, weakly unimodal functions may contain several "basins of attraction", see Fig. 2(c). *Multimodal functions*, on the other hand, are defined as functions with multiple local optima such that not all local optima are also global optima, see Fig. 2(d).

Note that both weakly unimodal and multimodal functions can be decomposed into several strictly unimodal functions, each of which has their own "basin of attraction". The latter can be defined as an interval, a subset of the domain, where f is strictly unimodal. For example, in Figs. 2(c) and 2(d) we have two basins of attractions $[-10, 0]$ and $[0, 10]$ (boundary points can be part of two basins of attractions). Our analyses of AVM variants finding global optima on strictly unimodal functions carry over to finding local optima on weakly unimodal and multimodal functions under the assumption that AVM never leaves the starting point's basin of attraction. Fig. 2(c) gives an example of a landscape where this assumption holds for almost all starting points.

4. Analysis of the original AVM

4.1. Original AVM with IPS

The original AVM due to Korel [13] uses the following local search, shown in Algorithm 2, that we name *Iterated Pattern Search* (IPS). Starting at x, IPS first evaluates points $x-1$ and $x+1$ to identify a gradient. Unless x is a local optimum, it then performs a so-called pattern search, moving in the direction of decreasing f-values. The step size doubles with each step, so when the gradient is towards increasing indices IPS traverses the points

$$x,\ x+1,\ x+1+2,\ x+1+2+4,\ \dots,\ x+1+2+4+\cdots+2^j.$$

Since $\sum_{i=0}^{j} 2^j = 2^{j+1} - 1$ this sequence is equal to

$$x,\ x+2^1-1,\ x+2^2-1,\ x+2^3-1,\ \dots,\ x+2^{j+1}-1.$$

Pattern search stops if the next point does not improve the fitness, which happens on unimodal functions when the optimum is being overshot. This process is iterated; that is, IPS then starts another exploration. If the function is unimodal, IPS gets close to the optimum over time. However, this line search can be relatively slow. The reason is that IPS accelerates during exploration, but after overshooting the optimum IPS starts another exploration from scratch.

Algorithm 2 Iterated Pattern Search, starting at $x \in D$.

```
1: while true do
2:   if $f(x-1) \geq f(x)$ and $f(x+1) \geq f(x)$ return $x$
3:   if $f(x-1) < f(x+1)$ then let $k := -1$ else let $k := 1$
4:   while $f(x+k) < f(x)$ do
5:     let $x := x+k$, $k := 2k$
```

4.2. Upper bound for original AVM

We start our investigations with Iterated Pattern Search, the local search used in the original AVM [13]. An upper bound $O(\log^2 n)$ (for domain size n) was proven for AVM with IPS in [3], for the special case that there is a linear relationship between the function value and the distance to the optimum. The following statement holds for arbitrary strictly unimodal functions. Here and in the remainder log denotes the binary logarithm $\log_2$.

Theorem 1. *Consider Iterated Pattern Search on a strictly unimodal function $f: \mathbb{Z} \to \mathbb{R}$ where d denotes the initial distance from the starting point to the optimum. Then IPS finds an optimum after querying at most $(\log d)^2 + 8\log d + 4$ values. This also holds for functions f that result from a strictly unimodal function f' and assigning function values $\min(f')$ to further points as in (2), d being the initial distance to the optimum of f'.*

The last statement implies that we get an upper bound of order $O((\log d)^2)$ for many further common functions. Examples are functions where opt is a global optimum and all points $x > \text{opt}$ or $x < \text{opt}$ are global optima as well (see Fig. 2(b)). In particular, all functions considered in Arcuri's work [2] are covered by this statement. However, for functions with many global optima the upper bound may not be tight [2].

Proof of Theorem 1. We allow the algorithm to traverse points outside of D, but assume that all $x' \notin D$ are worse than all $x \in D$.

We consider *passes* of IPS, corresponding to one iteration of the outer while loop in Algorithm 2: a pass starts with an exploratory search examining the two neighbouring solutions of the current point (index ± 1). It then performs a pattern search, doubling the distance travelled in each step. Note that a pass starting with the optimum makes exactly 3 queries. If a pass queries points up to a distance of $\sum_{j=0}^{i} 2^j = 2^{i+1} - 1$ from the initial value, it queries $i + 3$ values.

Without loss of generality assume that the current position is 0 and the optimum is at d. If $d = 1$ we need 4 queries, hence we assume in the following that $d \geq 2$. We claim that within at most 2 passes the distance to the optimum has been reduced to at most $\lfloor d/2 \rfloor$. Let i be the unique integer such that $2^i - 1 \leq d < 2^{i+1} - 1$. Note that pattern search queries $2^i - 1$ and $2^{i+1} - 1$ as the points are strictly improving in $[0, 2^i - 1]$. We consider two cases. First assume that $2^i - 1$ is better than $2^{i+1} - 1$. Then pattern search stops at $2^i - 1$ and since $d \leq 2^{i+1} - 2$ the new distance to the optimum is at most $\lfloor d/2 \rfloor$. The number of queries made is at most $i + 3$, and since $d \geq 2^i - 1 \geq 2^{i-1}$ this is at most $\lfloor \log d \rfloor + 4$.

Now consider the case that $2^i - 1$ is worse than $2^{i+1} - 1$. This implies $2^i \leq d \leq 2^{i+1} - 2$. Then pattern search will query $2^{i+2} - 1$ and stop at $2^{i+1} - 1$ as $2^{i+2} - 1$ is worse than $2^{i+1} - 1$. The second pass will traverse positions $2^{i+1} - 2^1$, $2^{i+1} - 2^2, 2^{i+1} - 2^3 \ldots$. Since by unimodality all points in $[0, 2^i]$ are increasingly better, pattern search will stop at some $2^{i+1} - 2^j$ with $0 \leq j \leq i$. As the optimum must be within $[\max(2^i, 2^{i+1} - 2^{j+1}), 2^{i+1} - 2^{j-1}]$, and the new current point is $2^{i+1} - 2^j$, the distance between the current point and the optimum has decreased to at most 2^j for $j < i$ and 2^{i-1} for $j = i$. In both cases the new distance is bounded by $\lfloor d/2 \rfloor$.

In the worst case, we need one pass querying up to $i + 4$ values, and a second pass querying up to $i + 3$ points. The total is $2i + 7 \leq 2\lfloor \log d \rfloor + 7$ as $d \geq 2^i$.

The total number of queries made, $T(d)$, is then subject to the following recurrence: $T(0) = 3$, $T(1) = 4$, and $T(d) \leq 2\lfloor \log d \rfloor + 7 + T(\lfloor d/2 \rfloor)$ for $d > 1$. The floor functions in this formula imply that, when repeatedly expanding terms $T(\cdot)$, we get the same recurrence for $T(d)$ as for $T(2^{\lfloor \log d \rfloor})$. Solving the latter gives

$$\begin{aligned} T(d) &\leq \sum_{k=1}^{\lfloor \log d \rfloor} (2k + 7) + T(1) \\ &\leq 2 \cdot \frac{\lfloor \log d \rfloor (\lfloor \log d \rfloor + 1)}{2} + 7\lfloor \log d \rfloor + T(1) \\ &\leq (\log d)^2 + 8 \log d + 4. \end{aligned}$$

The last remark of the statement holds true since adding further global optima can only decrease the expected time until some global optimum is found. □

4.3. Original AVM is slow in the worst case

The following result shows that the upper bound from Theorem 1 is asymptotically tight. Both bounds together show that the worst-case running time of IPS is of order $\Theta((\log d)^2)$, when initial points up to distance d are allowed. Note that the result applies to *every* unimodal function, hence AVM always has a bad worst-case performance.

Theorem 2. *Consider Iterated Pattern Search minimising an arbitrary strictly unimodal function $f : \mathbb{Z} \to \mathbb{R}$. If there are feasible starting points with distances $0, 1, \ldots, d$ to the optimum, the worst case number of unique fitness evaluations is at least*

$$\frac{(\log d)^2}{10} - O(\log d).$$

Proof. We can assume w.l.o.g. that the optimum is at position 0. If the domain D is bounded on one side, we consider an extended function f' where the domain is $\mathbb{Z}$ and $f(x) = \infty$ for all $x \notin D$.

The challenge in proving the statement is that it holds for an arbitrary unimodal function f. We do not know any details of this unimodal function; we only know from (1) that f is monotonically decreasing in the negative range and monotonically increasing in the positive range. This shape implies that when IPS overshoots the optimum by travelling (w.l.o.g. towards increasing indices) from a point $\ell < 0$ towards $r > 0$, the comparison of $f(\ell)$ and $f(r)$ determines whether a pass of IPS will stop at ℓ or r, triggering the next pass from there. Note that these are the only two possibilities due to (1). Note that the distance to the optimum for ℓ and r is no reliable indicator for the comparison of $f(\ell)$ and $f(r)$: even if, say, ℓ is much further away, $|\ell| \gg r$, there are many unimodal functions f that still satisfy $f(\ell) < f(r)$, namely if the slope of f in the positive range is steeper than that of f in the negative range.

The main idea of the proof is to consider two worst-case starting points within a range that grows with a parameter i: a point $\ell_i < 0$ left of the optimum and another point $r_i > 0$ right of the optimum. Having a worst-case point on either side of the optimum is necessary as IPS may stop a pass on either side of the optimum. We provide a lower bound for the fastest time starting from either ℓ_i or r_i. Then we consider a larger range for parameter $i+1$ and determine worst-case points ℓ_{i+1} and r_{i+1}. These points are chosen based on ℓ_i, r_i and the comparison of $f(\ell_i)$ and $f(r_i)$, in such a way that IPS starting from either ℓ_{i+1} or r_{i+1} will stop a pass at the better search point among ℓ_i and r_i. As the above-mentioned lower bound applies to the fastest time from either ℓ_i or r_i, this allows us to add this lower bound to the time it takes to travel from ℓ_{i+1} or r_{i+1} to ℓ_i or r_i. We get a new lower bound for the fastest time from either ℓ_{i+1} or r_{i+1}. The claimed lower bound then follows by induction over i.

Formalising this idea, we define $T_s(\ell, r)$ as the number of different search points evaluated when IPS starts in s, counting evaluations from the set $\{\ell, \dots, r\}$ only. If $s \notin D$ we let $T_s(\ell, r) := \infty$. Let $T(\ell, r) := \min\{T_\ell(\ell, r), T_r(\ell, r)\}$ be the smallest such number when starting from either ℓ or r.

Define $\ell_0 = r_0 = 0$ and $T(\ell_0, r_0) = 1$. Assume we have $\ell_i \le 0 \le r_i$ for some $i \in \mathbb{N}_0$, such that ℓ_i and r_i are not both outside f's domain. Let $\Delta_i := 2^{\lfloor \log(r_i - \ell_i) \rfloor + 1}$ for $i \in \mathbb{N}$ and $\Delta_0 := 1$ be the smallest power of 2 such that $\Delta_i > r_i - \ell_i$.

We define new points ℓ_{i+1}, r_{i+1} according to the following case distinction. First assume $f(\ell_i) > f(r_i)$, which implies $f(x) > f(r_i)$ for all $x \le \ell_i$. It also implies that r_i exists. Let $r_{i+1} := r_i + \Delta_i - 1$ and $\ell_{i+1} := r_i - 2\Delta_i + 1$. If IPS starts at r_{i+1}, it will sample points at

$$r_{i+1}, r_{i+1} - \left(2^1 - 1\right), \dots, r_{i+1} - \left(2^{\log(\Delta_i)} - 1\right) = r_i, r_i - \Delta_i$$

and since the fitness improves in every step but the last one, IPS will stop at r_i and restart exploration from there. All points but $r_i - \Delta_i$ are guaranteed to exist and are contained in $\{\ell_{i+1}, \dots, r_{i+1}\}$; so IPS evaluates $\Delta_i + 1$ different search points from that set before restarting exploration. From r_i IPS needs time at least $T(\ell_i, r_i) - 1$ since so far IPS has evaluated a single search point from the set $\{\ell_i, \dots, r_i\}$, namely r_i. We thus have established the recurrence

$$T_{\ell_{i+1}}(\ell_{i+1}, r_{i+1}) \ge \Delta_i + T(\ell_i, r_i). \tag{3}$$

Similarly, if ℓ_{i+1} exists and IPS starts from there, it will sample points at

$$\ell_{i+1}, \ell_{i+1} + 2^1 - 1, \dots, \ell_{i+1} + 2^{\log(\Delta_i)} - 1 = r_i - \Delta_i, r_i, r_i + 2\Delta_i.$$

The fitness improves in each step but the last one, and so IPS will stop at r_i and start exploration from there. Not counting the evaluation of $r_i + 2\Delta_i$, IPS evaluates $\Delta_i + 2$ search points, hence as above we get

$$T_{r_{i+1}}(\ell_{i+1}, r_{i+1}) \ge \Delta_i + T(\ell_i, r_i) + 1. \tag{4}$$

Putting (3) and (4) together, we have shown

$$T(\ell_{i+1}, r_{i+1}) \ge \Delta_i + T(\ell_i, r_i).$$

This also holds when $\ell_{i+1} \notin D$ as then $T_{\ell_{i+1}}(\ell_i, r_i) = \infty$. The case $f(\ell_i) < f(r_i)$ is symmetric, and if $f(\ell_i) = f(r_i)$ we also get the same recurrence as IPS stops at r_i as in the case $f(\ell_i) > f(r_i)$ when starting from ℓ_{i+1} and IPS stops at ℓ_i as on the other case when starting from r_{i+1}.

It follows that

$$T(\ell_i, r_i) \ge \sum_{j=1}^{i-1} \Delta_j + T(\ell_0, r_0) \ge 1 + \sum_{j=1}^{i-1} \log(r_j - \ell_j).$$

Note that for $i \ge 1$ the difference $r_{i+1} - \ell_{i+1}$ does not depend on whether $f(\ell_i) > f(r_i)$ or not, hence w.l.o.g. we use the definition of ℓ_{i+1}, r_{i+1} from the case $f(\ell_i) > f(r_i)$. Along with $\Delta_i \ge r_i - \ell_i + 1$ we get

$$r_{i+1} - \ell_{i+1} = r_i + \Delta_i - 1 - (r_i - 2\Delta_i + 1) = 3\Delta_i - 2 > 3(r_i - \ell_i).$$

Expanding and using $r_1 - \ell_1 = 2$ yields

$$r_{i+1} - \ell_{i+1} > 2 \cdot 3^i.$$

Thus,

$$\begin{aligned} T(\ell_i, r_i) &\ge 1 + \sum_{j=1}^{i-1} \log\left(2 \cdot 3^{j-1}\right) \\ &= i + \sum_{j=0}^{i-2} \log\left(3^j\right) \\ &= i + \log(3) \cdot \frac{(i-2)(i-1)}{2} > \frac{\log(3)}{2} \cdot i^2 - O(i). \end{aligned}$$

It is easy to verify by induction that $r_i \le 7^{i-1}$ and $|\ell_i| \le 7^{i-1}$ for all $i \in \mathbb{N}$. Putting $i := \lfloor \log_7(d) \rfloor + 1$ along with $i \ge \log_7(d) = \log_7(2) \cdot \log_2(d)$ then gives a lower bound of

$$\frac{\log(3)}{2} \cdot \left(\log_7(2)\right)^2 \cdot (\log d)^2 - O(\log d) > \frac{(\log d)^2}{10} - O(\log d). \qquad \square$$

4.4. Original AVM is slow on average

The bad worst-case performance of IPS is not simply due to a few unlucky choices of the initial point. In fact, most starting points lead to a running time of order $\Theta((\log d)^2)$. To see this, we consider the specific function $f(x) = |x|$, which is equivalent to the normalised function from Fig. 2(a). We show that when the starting point is chosen such that the distance between starting point and target is uniform in some interval, then we still get a lower bound of order $(\log d)^2$.

Note that $f(x) = |x|$ is quite an easy function as points closer to the optimum 0 are better than points that are further away from it. This encourages IPS to stop at the closest point to the optimum traversed in a pattern search, but we still get a time of $\Omega((\log d)^2)$.

Theorem 3. *Consider Iterated Pattern Search minimising the function $f(x) = |x|$ such that the starting point is chosen uniformly at random from $\{-2^i, \dots, 2^i - 1\}$, for some $i \in \mathbb{N}_0$. The expected number of unique fitness evaluations is at least $\frac{i^2}{6}$.*

Note that the choice of the initial starting point corresponds to choosing a random integer in the two's complement representation on $i+1$-bit words, the standard method for representing signed integers in programming languages.

Proof of Theorem 3. Let $T(i)$ denote the expected number of different search points queried when the starting point is chosen uniformly at random from $\{-2^i, \dots, 2^i - 1\}$. The claim $T(i) \ge i^2/6$ is trivial for $i = 0$ and $i = 1$.

If IPS starts at some value $x < 0$ (the case $x > 0$ is symmetric), IPS will start a pattern search exploring points with higher indices, querying points at $x_1 := x + 2^0$, $x_2 := x + 2^0 + 2^1$, $x_3 := x + 2^0 + 2^1 + 2^2$, etc. (We do not count a potential evaluation of the point at $x - 1$ since it might not be in the domain of feasible search points.) Let $x_j := x + \sum_{\ell=0}^{j-1} 2^\ell = x + 2^j - 1$ for $1 \le j \le i$ be the first search point queried where $x_j \ge 0$. Now IPS will stop pattern search and continue with either x_{j-1} or x_j, depending on which is better. If x_j is better, IPS will also query x_{j+1}; but as this might be out of range, we do not count a potential evaluation of x_{j+1}.

Due to the fitness function used, the point with the smaller absolute value from either x_{j-1} or x_j is better. Note that their index difference is $x_j - x_{j-1} = 2^{j-1}$, so $x_j \in \{0, \dots, 2^{j-1} - 1\}$. If $x_j \in \{0, \dots, 2^{j-2} - 1\}$, x_j is better than x_{j-1}, and IPS starts another pass at $\{0, \dots, 2^{j-2} - 1\}$. Otherwise, x_{j-1} is better and IPS will start another pass at $\{-2^{j-2}, -2^{j-2} + 1, \dots, -1\}$. All these positions are attained with the same probability, hence we are in the same setting as described in the statement, with $j - 2$ in place of i.

The probability of stopping at x_j being the first point where $x_j \ge 0$ ($x_j \le 0$ when starting at $x > 0$), for $1 \le j \le i$, is $2 \cdot 2^{j-1}/2^{i+1} = 2^{j-i-1}$ as there are $x_j - x_{j-1} = 2^{j-1}$ positions for x where this happens when $x < 0$ and the same holds for $x > 0$. Recall that all initial positions are chosen uniformly at random, and there are 2^{i+1} feasible positions.

While getting to x_j IPS has queried at least $j + 1$ mutually different points $x, x_1, \dots, x_j$. Then the remaining time is at least $T(j-2) - 1$; the reason for subtracting 1 is that we have already queried x_j. Defining $T(-1) := 0$, we have established the following recurrence

$$\begin{aligned}
T(i) &\ge \sum_{j=1}^{i} 2^{j-i-1} \cdot \left(j + 1 + T(j-2) - 1\right) \\
&= \sum_{j=1}^{i} j \cdot 2^{j-i-1} + \sum_{j=1}^{i} 2^{j-i-1} \cdot T(j-2) \\
&= \sum_{j=0}^{i-1} (j+1) \cdot 2^{j-i} + \sum_{j=-1}^{i-2} 2^{j-i+1} \cdot T(j) \\
&= \sum_{j=0}^{i-1} (j+1) \cdot 2^{j-i} + \sum_{j=0}^{i-2} 2^{j-i+1} \cdot T(j)
\end{aligned}$$

as $T(-1) = 0$. Along with $\sum_{j=0}^{i-1} (j+1) \cdot 2^{j-i} = i - 1 + 2^{-i}$, we get

$$T(i) \ge i - 1 + 2^{-i} + \sum_{j=0}^{i-2} 2^{j-i+1} \cdot T(j).$$

Assume for an induction that $T(j) \geq j^2/6$ all $0 \leq j < i$. Then

$$\begin{aligned} T(i) &\geq i - 1 + 2^{-i} + \sum_{j=0}^{i-1} 2^{j-i} \cdot \frac{j^2}{6} \\ &= i - 1 + 2^{-i} + \frac{1}{6} \cdot \sum_{j=0}^{i-1} 2^{j-i} \cdot j^2 \\ &= i - 1 + 2^{-i} + \frac{1}{6} \cdot \left(-4i + 6 + i^2 - 6 \cdot 2^{-i}\right) \\ &= \frac{i^2}{6} + \frac{i}{3} \end{aligned}$$

which implies the claim. □

5. Alternative local searches for the AVM

We now show that other local searches used in the framework provided by AVM only require $\Theta(\log d)$ evaluations instead of $\Theta((\log d)^2)$. This yields significant speedups over AVM's original local search method IPS, if the initial distance d to the optimum is not very small.

Our results formally only hold for unimodal functions, but they also indicate more generally that the alternate local searches converge faster to local optima. The reason is that the basin of attraction around a local optimum has the properties of a unimodal function. So, our analysis is applicable whenever the search does not leave the initial basin of attraction. It further can be used to estimate the remaining running time after the search has reached a certain basin of attraction as above.

5.1. AVM with geometric search

We propose to use more clever local searches that locate the optimum of a unimodal function more efficiently after the first exploration. The following *Geometric Search* uses a variant of binary search. The idea is to perform a pattern search, and then to use binary search to home in on the target. Thereby we are using the following fact: if pattern search queries search points x_{j-1}, x_j, x_{j+1}, stopping at x_j, we know that $f(x_{j-1}) > f(x_j) \leq f(x_{j+1})$. This implies that, if f is unimodal, the global minimum must lie in the set $\{x_{j-1}, \ldots, x_{j+1}\}$. We call it "Geometric Search" since the initial pattern search is performed with a geometric sequence of numbers.

Algorithm 3 Geometric Search, starting at $x \in D$.

```
1: if $f(x-1) \geq f(x)$ and $f(x+1) \geq f(x)$ return $x$
2: if $f(x-1) < f(x+1)$ then let $k := -1$ else let $k := 1$
3: while $f(x+k) < f(x)$ do
4:     let $x := x+k$, $k := 2k$
5: let $\ell := \min(x - k/2, x+k)$, $r := \max(x - k/2, x+k)$
6: while $\ell < r$ do
7:     if $f(\lfloor(\ell+r)/2\rfloor) < f(\lfloor(\ell+r)/2\rfloor + 1)$ then
8:         $r := \lfloor(\ell+r)/2\rfloor$
9:     else
10:        $\ell := \lfloor(\ell+r)/2\rfloor + 1$
11: return $\ell$
```

The following result shows that AVM with Geometric Search finds the optimum of any unimodal function in time logarithmic in the initial distance.

Theorem 4. *Consider a one-dimensional search on a unimodal function $f: \mathbb{Z} \to \mathbb{R}$ where $d \geq 1$ denotes the initial distance from the starting point to the optimum. Then Geometric Search finds an optimum after querying at most $3\log d + 5$ search points.*

Proof. Let i be such that $2^i \leq d < 2^{i+1}$. By the same arguments as in the proof of Theorem 1 pattern search stops at either $2^i - 1$ or $2^{i+1} - 1$ after querying at most $i + 3$ points. We pessimistically assume that it stops at $2^{i+1} - 1$, which results in the algorithm putting $\ell := 2^i - 1$ and $r := 2^{i+2} - 1$. We claim that each iteration of binary search updates ℓ, r towards ℓ', r' such that $r' - \ell' \leq \lfloor(r - \ell)/2\rfloor$. If $r' = \lfloor(\ell + r)/2\rfloor$ we have

$$r' - \ell' = \left\lfloor \frac{\ell + r}{2} \right\rfloor - \ell = \left\lfloor \frac{r - \ell}{2} \right\rfloor.$$

Otherwise, $\ell' = \lfloor \frac{\ell+r}{2} \rfloor + 1$ and using $-\lfloor x \rfloor - 1 \le \lfloor -x \rfloor$ for $x \in \mathbb{R}$ we get

$$r' - \ell' = r - \left\lfloor \frac{\ell+r}{2} \right\rfloor - 1 \le r + \left\lfloor -\frac{\ell+r}{2} \right\rfloor = \left\lfloor \frac{r-\ell}{2} \right\rfloor.$$

Initially $r - \ell < 2^{i+2}$, and due to the floor functions we get the same recurrence as for 2^{i+1}. Two queries are needed to replace the current distance by its floored half, ending at 0 with no further queries. Hence we need an additional amount of $2i + 2$ queries, leading to a total of $3i + 5 \le 3 \log d + 5$ queries. □

It is easy to see that, in the setting of Theorem 4, Geometric Search always needs $3 \log d \pm O(1)$ queries: the initial pattern search queries $\log d \pm O(1)$ points before proceeding to an elimination search on an interval of length $\Omega(d)$. Geometric Search only stops after this interval has been reduced to a single point. Since halving this interval makes 2 queries, this process requires $2 \log d \pm O(1)$ further queries.

5.2. AVM with Lattice Search

Lattice Search [23, Section 8.2] is a refinement of *Fibonacci Search* [12] for integer domains. The idea of Fibonacci search is to evaluate search points according to Fibonacci numbers $F_1 = 1, F_2 = 1$, and $F_{n+2} = F_n + F_{n+1}$ for $n \in \mathbb{N}$ in such a way that only one new search point needs to be evaluated in each iteration. Assume we know that the optimum is in some interval $[\ell, \ell + F_n]$. Fibonacci search would then compare $\ell + F_{n-2}$ and $\ell + F_{n-1}$. If $f(\ell + F_{n-2}) \le f(\ell + F_{n-1})$ we know that an optimum must be in the interval $[\ell, \ell + F_{n-1}]$, and we iterate by comparing $\ell + F_{n-3}$ and $\ell + F_{n-2}$. The latter point has already been evaluated, hence only one new evaluation is required. Likewise, if $f(\ell + F_{n-2}) > f(\ell + F_{n-1})$ then an optimum must lie in $[\ell + F_{n-2}, F_n]$ and we iterate by setting $\ell' := \ell + F_{n-2}$ and evaluating $\ell' + F_{n-3}$ and $\ell' + F_{n-2}$. Since $\ell' + F_{n-3} = \ell + F_{n-1}$, it has already been evaluated, again leaving just one new evaluation.

In terms of unique evaluations, Fibonacci search is faster than Geometric search. Kiefer [12] showed that Fibonacci Search in continuous domains is an optimal search technique in the following sense. Consider all search techniques using a fixed number of function evaluations. Then Fibonacci search has the largest possible interval of input values for which it guarantees to find a solution whose quality differs by at most $\varepsilon > 0$ from that of an optimum.

Lattice Search further refines this principle by exploiting that for integer domains in the case $f(\ell + F_{n-2}) \le f(\ell + F_{n-1})$ the last point $\ell + F_{n-1}$ cannot be the unique optimum, hence the interval reduces by one further point to $[\ell, \ell + F_{n-1} - 1]$. Likewise, in the other case the interval becomes $[\ell + F_{n-2} + 1, F_n]$. In both cases the interval is reduced by 1 point, compared to Fibonacci search. These gains accumulate in each iteration, and Lattice Search on integers needs even fewer function evaluations than Fibonacci Search on a comparable continuous domain.

It is known that Lattice search can find the minimum of a unimodal function on a domain of integers $\{1, \dots, F_n - 1\}$ using $n - 2$ function evaluations [23, p. 190] (note that Monahan [23, p. 190] uses the definition $F_0 = 1$, $F_1 = 1$, $F_2 = 2$, $\dots$).

The local search using Lattice Search is shown in Algorithm 4.

Algorithm 4 Lattice Search, starting at $x \in D$.

```
1: if $f(x-1) \ge f(x)$ and $f(x+1) \ge f(x)$ return $x$
2: if $f(x-1) < f(x+1)$ thenlet $k := -1$ else let $k := 1$
3: while $f(x+k) < f(x)$ do
4:     let $x := x + k$, $k := 2k$
5: let $\ell := \min(x - k/2, x + k)$, $r := \max(x - k/2, x + k)$
6: let $n := \min\{n \mid F_n \ge r - l + 2\}$
7: while $n > 3$ do
8:     if $\ell + F_{n-1} - 1 \le r$ and $f(\ell + F_{n-2} - 1) \ge f(\ell + F_{n-1} - 1)$ then
9:         let $\ell := \ell + F_{n-2}$
10:    let $n := n - 1$
11: return $\ell$
```

Note that the initial pattern search is done in a geometric fashion, i.e., increasing the step size geometrically as in IPS and Geometric Search. There is a related search technique called *Fibonaccian Searching* [7] (not to be confused with Fibonacci Search) where pattern search is done by means of Fibonacci numbers. The reason that we are using geometric pattern search is that it is generally faster.

Lattice Search further improves the leading constant preceding the $\log d$ term; using Fibonacci numbers to search for the optimum in the interval identified by pattern search is more efficient than the binary search used in our Geometric Search procedure.

Theorem 5. *Consider a one-dimensional search on a unimodal function $f: \mathbb{Z} \to \mathbb{R}$ where $d \ge 1$ denotes the initial distance from the starting point to the optimum. Then Lattice Search finds an optimum after querying at most $2.45 \log d + 7$ search points.*

Proof. As in the previous proofs, pattern search stops after querying at most $\lceil \log d \rceil + 3 \le \log d + 4$ points. Afterwards, lattice search is run on an interval of size at most $4d$.

Let F_m be the smallest Fibonacci number such that $F_m - 2 \ge 4d$. We know that Lattice Search on a set $\{1, \dots, F_m - 1\}$ uses $m - 2$ function evaluations [23, p. 190] with our definition of F_m. This corresponds to our setting after a trivial index transformation.

Using a well-known closed formula for F_m,

$$F_m = \frac{(1+\sqrt{5})^m - (1-\sqrt{5})^m}{2^m \cdot \sqrt{5}},$$

we get for odd m, using $(1-\sqrt{5})^m < 0$,

$$F_m \ge \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^m.$$

For even m, we clearly have

$$F_m \ge F_{m-1} \ge \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^{m-1} = \frac{2}{\sqrt{5}+5} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^m.$$

Therefore, the required inequality $F_m \ge 4d + 2$ is implied by

$$\left(\frac{1+\sqrt{5}}{2}\right)^m \ge (4d+2) \cdot \frac{\sqrt{5}+5}{2}$$

$$\Leftrightarrow \quad m \ge \frac{\log((4d+2) \cdot \frac{\sqrt{5}+5}{2})}{\log(\frac{1+\sqrt{5}}{2})}.$$

The last term on the right-hand side is upper-bounded by $1.45 \log d + 5$. It follows that lattice search requires at most $m - 2 \le 1.45 \log d + 3$ evaluations. Along with the time for pattern search this proves the claim. □

6. Experiments

We now provide an empirical comparison of the AVM using our alternative local searches to complement our theoretical results. We first show results on simple functions, designed to provide additional empirical evidence about the better scalability of the alternative local searches. However, real-world programs are not so straightforward, involving fitness landscapes of varying shapes. Therefore, following some simple initial experiments, we continue onto performing experiments with five real-world programs.

6.1. Experiments with unimodal functions

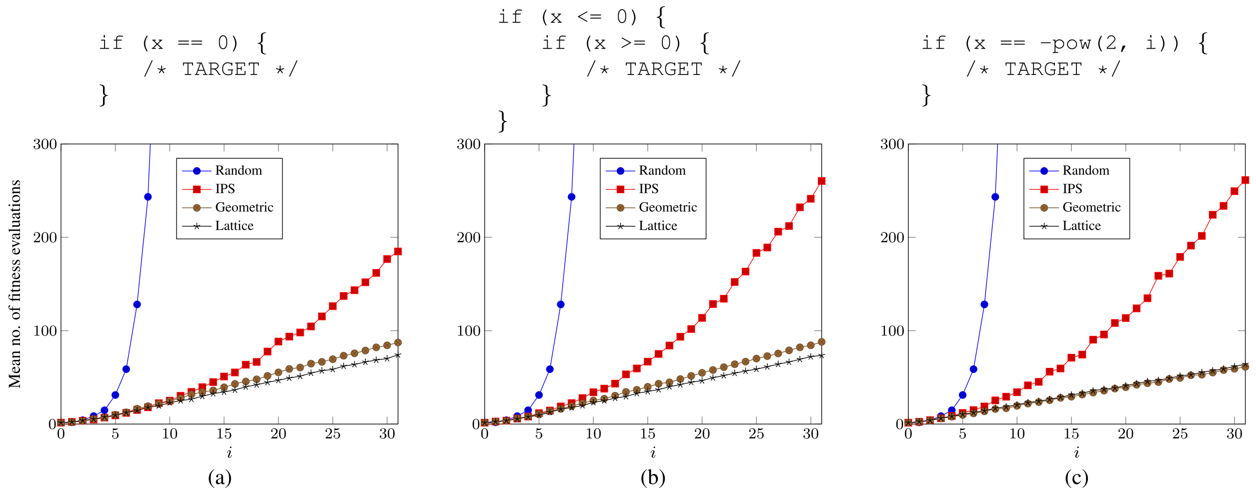
We return to the simple and illustrative problem from Fig. 2(a) with just a single variable, recalling that this corresponds to the following program:

```
void is_zero(int x) {
  if (x == 0) { /* TARGET BRANCH */ }
}
```

The goal is to find an input x that is equal to zero. The approach level is zero (since the branching condition is always reached), and the branch distance is norm$(|x|)$. Because the local searches do not rely on comparing absolute values, but instead their corresponding ranks, this problem is equivalent to minimising the fitness function $f(x) = |x|$ (where the branch distance is not normalised) as analysed in Theorem 3.

Studying this simple program allows us to isolate the impact of the range and the initial distance d from the optimum on the running time of the AVM. Our theoretical results show that, when the starting point is chosen uniformly at random from a set $\{-d, \dots, d-1\}$ for d a power of 2, the AVM with IPS will take $\Theta((\log d)^2)$ steps whereas the AVM with Geometric Search or Lattice Search will succeed in only $O(\log d)$ steps. Our theoretical results also give bounds with precise constants and precise small-order terms; experiments can reveal further insights into how tight these bounds are.

In order to obtain broader results, we also include experiments on two other strictly unimodal functions. Recall that for any strictly unimodal function the function values increase when moving away from the optimum in both directions. Because we only consider searches that compare ranks of search points rather than their absolute function values, the characteristics that define a strictly unimodal function may be viewed as the position of the optimum and the relative heights of the slopes on either side of the optimum.



```
if (x == 0) {
    /* TARGET */
}
```

```
if (x <= 0) {
    if (x >= 0) {
        /* TARGET */
    }
}
```

```
if (x == -pow(2, i)) {
    /* TARGET */
}
```

Fig. 3. Comparisons of mean number of unique fitness evaluations for three strictly unimodal target branches with various local searches. The domain is chosen as $\{-2^i, \dots, 2^i - 1\}$ for $i \in \{0, \dots, 31\}$. The first plot (a) corresponds to the function $f(x) = |x|$ from Fig. 2(a) and Theorem 3. The second branch (b) is similar, but all $x > 0$ have a worse fitness than all $x \le 0$. The third branch (c) shows a variant of (a) where the optimum is located at one end of the feasible domain.

In problem (a), i.e., $f(x) = |x|$, the optimum is located at zero with slopes of equal gradient to the left and right of the optimum. In problem (b) the optimum is also located at zero, but all search points to the right of the optimum are strictly worse that all points to the left of the optimum because they have a greater approach level. In problem (c) the optimum is located at the left end of the domain right next to the infeasible search space.

The performance of IPS, Geometric Search and Lattice Search was measured over the following ranges $[-d, d-1]$, where $d \in \{1, 2, 4, 8, \dots, 2^{31}\}$. For each range 100 runs of each local search were performed and the number of unique fitness evaluations to find the global optimum was counted. In each run the starting position x_1 was chosen uniformly at random from the corresponding range (including boundaries).

Fig. 3 shows how the average performance of each local search scales with increasing domain size. Here the variable i is related to the logarithm of the distance, i.e., $i = \log_2(d)$. We also included pure random search (marked as "Random" in the figure), for completeness.[3] Random search is commonly involved in empirical studies applying optimisation to software testing. This is because random search is used frequently in software testing as a technique for testing in its own right, and is referred to as "random testing". Random testing is often the baseline on which we want to improve—that is, we are comparing with the simplest technique that could be applied. Unsurprisingly, it frequently fails to find the desired input, as its expected running time equals the input domain size.

The empirical results agree with our theoretical results as one can clearly see a different scaling behaviour between IPS and the two alternative local searches, Geometric Search and Lattice Search. For small d the performance is similar, but as d grows the differences become obvious. The performance of all local searches is similar across all three problems. It seems that problem (a), $f(x) = |x|$ from Theorem 3, is the least difficult problem for IPS among (a), (b), and (c). The reason might be on problem (a) that the fitness reflects the distance to the optimum. This implies that whenever IPS overshoots the optimum, it will start a new pattern search from the closest point to the optimum. This is not always the case for (b) and (c).

In order to investigate the growth curve of IPS on problem (a), a second order polynomial was fit to the average running times of IPS using a weighted non-linear regression and then the χ^2 test was used to assess the goodness of fit. The equation of the fitted curve is as follows: $T(i) = 0.169623 \cdot i^2 + 0.717115 \cdot i + 1.5189$, where T is the mean number of fitness evaluations. Note that the leading constant 0.169623 almost exactly matches the constant $1/6 = 0.166666\dots$ from Theorem 3. For the fit, the obtained value of χ^2 is 27.6887 and the number of degrees of freedom is $32 - 3 = 29$ (i.e., the number of values of i minus the number of fitted parameters). Since $P(\chi^2 > 27.6887) = 0.5345$, this suggests that the fit is of high quality and explains the experimental results very well. We also applied the same fitting technique and assumed the same form of equation (i.e. a quadratic function) to investigate the growth curves for IPS on problems (b) and (c). Using the same notation as before, the equation for the fitted curve for problem (b) is $T(i) = 0.249093 \cdot i^2 + 0.682377 \cdot i + 1.5213$ with $\chi^2 = 26.0034$ and $P(\chi^2 > 26.0034) = 0.6253$ and that for problem (c) is $T(i) = 0.247606 \cdot i^2 + 0.824655 \cdot i + 1.4856$ with $\chi^2 = 30.4144$ and $P(\chi^2 > 30.4144) = 0.3935$. Similar to problem (a), the values of χ^2 for the curves in problems (b) and (c) indicate that a quadratic fit is the correct choice for approximating how the behaviour of IPS scales on both of these

[3] Fig. 3 shows the same data for Random in (a), (b), and (c) as random search behaves equally on every strictly unimodal function with the same domain size.

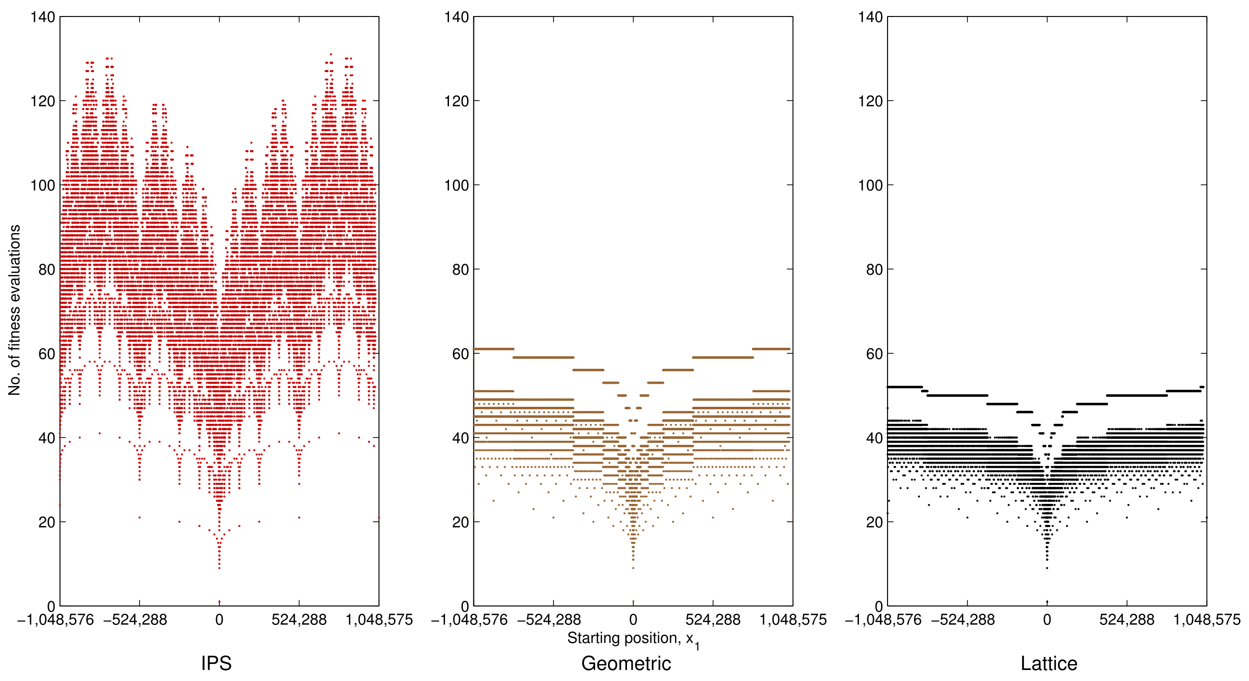


Fig. 4. Number of unique fitness evaluations for optimizing $f(x) = |x|$ with different local searches for each starting position $x_1 \in [-2^{20}, 2^{20} - 1]$. For the sake of clarity, the plots do not show all 2^{21} data points, but were downsampled to a selection of 25 000 data points for each search to give a good and accurate visual representation of all 2^{21} points, using the Largest-Triangle-Three-Buckets algorithm [27].

problems. The leading constants for the fitted curves of IPS on problems (b) and (c) are very similar but are noticeably greater than that of problem (a) suggesting that these two problems are more difficult for IPS to solve because on average require a greater number of fitness evaluations for a given domain size.

Along with the comparison of IPS on (a), (b), and (c), the empirical results indicate that the lower bound on the average-case performance from Theorem 3 may also hold for other strictly unimodal functions.

As all problems lead to similar results regarding the comparison of IPS and alternative local searches, the upcoming statistical evaluation of results just considers problem (a) studied in Theorem 3.

For each range the runtime distributions of the local searches in Fig. 3(a) were compared in a pairwise manner using the Mann–Whitney U test. In addition to p-values, the non-parametric Vargha–Delaney statistic $\hat{A}_{12}$ [30], which is computed from mean ranks, is reported as a measure of effect size. It is possible to interpret $\hat{A}_{12}$ as the probability that a run of the first search algorithm takes a larger number of fitness evaluations compared to that of the second search algorithm. The implication is that if $\hat{A}_{12} < 0.5$, then the first local search performs better overall whereas the opposite is true if $\hat{A}_{12} > 0.5$. Also, depending whether the absolute difference: $|\hat{A}_{12} - 0.5|$ is > 0.21, > 0.14, > 0.06, or ≤ 0.06, the corresponding effect size can be divided into the following categories: large, medium, small, and negligible.

The results show that for all ranges where $d \geq 2048$, IPS is worse than Geometric Search ($p \leq 0.0054$ and $\hat{A}_{12} \geq 0.61$) and also worse than Lattice Search ($p \leq 4.1 \times 10^{-8}$ and $\hat{A}_{12} \geq 0.72$). It was also found that for the same ranges Geometric Search is worse than Lattice Search ($p \leq 7.9 \times 10^{-8}$ and $\hat{A}_{12} \geq 0.72$).

In Fig. 4 we additionally show how the performance of Iterated Pattern Search on $f(x) = |x|$ depends on the precise choice of the starting point. Experiments were run for all initial values $x_1 \in [-2^{20}, 2^{20} - 1]$. The range was chosen large enough to reveal differences in the average performance between IPS and the alternative local searches (cf. Fig. 3(a) for $i = 20$). The drawback of this large range is that the amount of data was too large to be plotted as a whole. Attempting to plot all the data points would result in a figure that is difficult to render due to large amounts of visual overlap and high memory requirements. Instead, the approach we took was to plot a sample of 25 000 data points per search. We used a method called the Largest-Triangle-Three-Buckets [27] for down-sampling the data and deciding which points to draw. This method is designed to produce an accurate visual representation of the original data set by preserving as much detail as possible with regards to its overall shape and other defining features. It works by dividing the data set into the same number of 'buckets' as to be sampled, then choosing one point from each bucket such that the point which is selected forms the triangle with the greatest area when connecting to two other points from each of its own neighbouring buckets.

The performance of Iterated Pattern Search is symmetric around 0 (modulo tiny differences in tie-breaking) and the pattern looks like a fractal structure. This reflects the recursive nature of IPS, which also became visible in the recurrence arguments used in our proofs from Section 4. In accordance with our average-case analysis from Theorem 3, many starting points lead to rather high running times.

Table 2
Details of the test objects used in the experiments.

Function name	No. of branches	Inputs and ranges
clip_to_circle	42	$\{x_1 \ldots x_7\} \in [-2^{31}, 2^{31}-1]$
days_between	32	$\{x_1 \ldots x_6\} \in [-2^{15}, 2^{15}-1]$
gimp_rgb_to_hsv_int	14	$\{x_1 \ldots x_3\} \in [0, 255]$
gimp_hsv_to_rgb_int	16	$x_1 \in [0, 360]$, $\{x_2, x_3\} \in [0, 255]$
validate_card	10	$\{x_1 \ldots x_{16}\} \in [0, 9]$

The alternative local searches lead to smoother and smaller curves, as there the running time is always bounded by $O(\log d)$. The observable differences between Geometric Search and Lattice Search seem to be in line with the different leading constants in our upper running time bounds.

6.2. Experiments with real-world test objects

So far, we have been considering strictly unimodal functions, or variations thereof with additional global optima. This leaves open the question whether the alternative local searches also improve performance on branches with multimodal fitness landscapes. As discussed, our hope is that the alternative local searches are faster at finding local optima as multimodal functions consist of multiple unimodal functions. Moreover, if the function to be optimised by local search contains multiple local optima, replacing IPS by a different local search might lead to different local optima being returned. This can change the global search trajectory, with unforeseen effects; such a change can have a beneficial or a detrimental impact on performance, or no impact at all. This issue will be further discussed in Section 7; it corroborates the need for experiments on real-world test cases with multiple variables as it is not obvious whether faster local searches also improve global search performance.

In order to compare the performance of the local searches in a practical setting, we implemented the alternative AVM searches into and conducted our experiments with the IGUANA toolset [18], and selected five test objects, details of which are shown in Table 2. Each test object is written in C, and its source code was automatically instrumented by IGUANA for the purposes of collecting fitness information. The selected test cases cover variables from very different ranges, from just 10 values to 2^{32} in the full 32 bit integer range.

The *clip_to_circle* function is from the graphical front-end of the *SPICE* electronic circuit simulator. It clips a line—specified by two pairs of integer co-ordinates—to a circle, specified by a pair of integer co-ordinates and a further integer specifying its radius. The *days_between* function calculates the number of days between two dates. The source code for this function appears in reference [20]. The function takes six short integers as arguments, corresponding to the day, month and year of a pair of dates. We did not restrict these ranges to those corresponding to valid day and month numbers etc., since the function is designed to check for and correct invalid inputs. The functions *gimp_rgb_to_hsv_int* and *gimp_hsv_to_rgb_int* are colour space converters from the *GIMP* image editor. The former takes three integers in the range 0–255; specifying red, green, and blue colorspace components. The latter takes three arguments, the first an integer in the range 0–360 specifying a hue value (in degrees), followed by two further integers in the range 0–255 for saturation and value components. Finally, the *validate_card* function is an implementation of the Luhn algorithm for checking 16 digit credit card numbers, and takes 16 integers in the range 0–9 as arguments. Each test object consists of nested control structures in the form of *if* statements, *switch* statements and loops. IGUANA creates a pair of true and false branches when it detects that control flow diverges in the source code of a C function. Because there can be multiple pairs of branches in a single test object, each pair is labelled with the control flow graph number of the branching decision statement suffixed with a “T” or an “F” to distinguish the true branch from its false counterpart.

In the experiments, AVM with each local search was applied to each function 100 times. Each run was treated independently such that the starting position was chosen uniformly at random from the entire search space. We recorded the mean number of unique fitness evaluations, excluding evaluations of infeasible search points with coordinates outside their domain, as these could be identified with little computational effort. Also for Geometric Search and Lattice Search, if during the elimination stage of the search both middle points fall outside the domain, a decision is made to favour the solution closest to the boundary so that the search is directed back to the feasible region of the search space. For each branch pairwise comparisons of the runtime distributions corresponding to different local searches were performed using the Mann–Whitney U test. The results include raw p-values and non-parametric effect sizes, $\hat{A}_{12}$.

Only branches where random search performed notably worse than at least one of the local searches (i.e., $p < 0.05$ (significant) and $\hat{A}_{12} > 0.56$ (at least small effect size)) are considered. The purpose of filtering was to remove branches that are covered easily by any search as there is no reason to design a better algorithm for these branches. Similarly, there were 11 branches which were either infeasible (i.e., no inputs from the input domain execute it) or so hard that none of the tested algorithms found an optimum. In total, 28 out of 114 branches satisfied the selection criteria.

In order to get a more detailed insight into the structure of these remaining branches, we used a sampling approach to investigate how many fitness landscapes for line searches are unimodal. To this end, for each branch with variables $x_1, \ldots, x_n$ we picked an index $1 \le i \le n$ uniformly at random. All variables x_j, $j \ne i$, were fixed to constants $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n$

Table 3
Results of the sampling process to estimate the degree of unimodality for each branch listed in Table 4.

Function	Branch	Strictly unimodal (%)	Weakly unimodal (%)	Multimodal (%)
clip_to_circle	7F	29	71	0
	10F	26	74	0
	55T	18	17	65
	55F	24	22	54
	57F	18	19	63
	61T	13	18	69
	66T	12	20	68
	68T	22	16	62
	68F	11	14	75
	70F	19	16	65
	74T	15	18	67
	Overall %	*18.8*	*27.7*	*53.5*
days_between	28T	50	33	17
	32T	81	0	19
	34T	80	0	20
	34F	89	0	11
	Overall %	*75*	*8.3*	*16.8*
gimp_rgb_to_hsv_int	14T	0	100	0
	17T	0	100	0
	Overall %	*0*	*100*	*0*
gimp_hsv_to_rgb_int	4T	28	72	0
	11T	34	66	0
	Overall %	*31*	*69*	*0*
validate_card	5F	5	93	2
	7T	8	92	0
	7F	6	94	0
	9T	7	93	0
	9F	6	94	0
	11T	4	96	0
	11F	7	93	0
	14T	7	91	2
	14F	1	99	0
	Overall %	*5.7*	*93.9*	*0.4*
Overall %		**22.1**	**54.3**	**23.5**

chosen uniformly at random. This led to a line search problem $(c_1, \ldots, c_{i-1}, x_i, c_{i+1}, \ldots, c_n)$ with x_i being the only variable and the other coordinates being fixed. Then we scanned the whole range of x_i to classify the landscape as strictly unimodal, weakly unimodal, or multimodal. The domains of the input variables for *clip_to_circle* and *days_between* were too large to allow for a complete scan. So for these branches we sampled from a scaled version of that branch instead, where the range for all such variables was reduced to $[-2^7, 2^7-1]$. The sampling process was repeated 100 times for each branch, to yield an estimate of its degree of unimodality. Table 3 records the results. Across all 28 branches listed in Table 4, 22.1% of landscapes were strictly unimodal. A further 54.3% were weakly unimodal. Only 23.5% of all sampled settings were truly multimodal, i.e., contained local optima, which were not globally optimal. The largest proportion of these belong to *clip_to_circle*, with a smaller number also appearing in *days_between*.

The results of comparisons among AVM variants with Mann–Whitney U tests are recorded in Table 4, where we indicate statistical significance on a low level of 0.01. Wherever we detected a statistical significance, we compared mean ranks to identify which algorithm performed better.

AVM with Geometric Search performed significantly better than AVM with IPS on 12 branches, with small to large effect sizes. Geometric Search performed significantly worse than IPS on 3 branches. On strictly unimodal functions Lattice Search led to a better leading constant in our theoretical upper bounds and our previous experiments (Fig. 3), hence we expect AVM with Lattice Search to be slightly faster than AVM with Geometric Search. Indeed, using Lattice Search within AVM was significantly faster than AVM with IPS on 16 of the 28 branches, with a medium to large effect size. On all branches with domain sizes at least 2^{16}, AVM with Lattice Search was significantly faster than AVM with IPS.

Looking at the direct comparison between Geometric Search and Lattice Search, 8 branches showed a significant difference between the two searches, and on 7 of them Lattice Search was faster than Geometric Search. This notably includes branch 10F of *clip_to_circle*, where Lattice Search showed a slightly higher mean, but a smaller mean rank than Geometric Search. The effect sizes for comparisons of Geometric and Lattice Search were typically smaller than for comparisons with IPS.

Table 4
Results of test case experiments: mean numbers of unique fitness evaluations and the results of statistical evaluations. The p-values formatted in bold indicate significance at the 0.01 level. Similarly, effect sizes that are large, medium, small, and negligible are distinguished by bold, underlined, italic, and normal formatting respectively.

Function	Mean no. of fitness evaluations				IPS v geometric		IPS v lattice		Geometric v lattice	
	Branch	IPS	Geometric	Lattice	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
clip_to_circle	7F	853.60	300.88	283.86	**5.6×10^{-19}**	**0.86**	**1.5×10^{-19}**	**0.87**	**5.7×10^{-3}**	*0.61*
	10F	819.92	291.97	298.06	**1.1×10^{-20}**	**0.88**	**9.2×10^{-19}**	**0.86**	**5.4×10^{-3}**	*0.61*
	55T	1 988.89	1 250.49	1 087.02	**7.1×10^{-5}**	0.66	**4.8×10^{-6}**	0.69	7.0×10^{-1}	0.52
	55F	654.43	294.18	315.61	**6.7×10^{-14}**	**0.81**	**1.5×10^{-11}**	**0.78**	3.6×10^{-1}	0.46
	57F	627.69	378.89	307.42	**1.3×10^{-8}**	**0.73**	**1.6×10^{-12}**	**0.79**	3.2×10^{-1}	0.54
	61T	583.47	280.79	237.88	**2.1×10^{-14}**	**0.81**	**3.0×10^{-16}**	**0.83**	2.1×10^{-1}	0.55
	66T	788.99	442.36	328.04	**2.6×10^{-7}**	**0.71**	**1.3×10^{-13}**	**0.80**	3.7×10^{-2}	*0.59*
	68T	2 588.73	1 973.70	1 282.51	4.1×10^{-2}	*0.58*	**6.5×10^{-6}**	0.68	2.2×10^{-2}	*0.59*
	68F	870.05	559.26	488.91	**4.6×10^{-3}**	*0.62*	**3.9×10^{-6}**	0.69	6.4×10^{-2}	*0.58*
	70F	978.58	496.15	511.57	**1.7×10^{-7}**	**0.71**	**1.1×10^{-6}**	0.70	5.4×10^{-1}	0.47
	74T	859.94	459.82	458.20	**7.4×10^{-7}**	0.70	**3.0×10^{-6}**	0.69	7.8×10^{-1}	0.49
days_between	28T	71.78	59.79	55.02	**3.0×10^{-7}**	0.71	**3.9×10^{-10}**	**0.76**	6.7×10^{-2}	*0.57*
	32T	102.43	106.52	83.00	3.0×10^{-1}	0.54	**3.2×10^{-5}**	0.67	**1.0×10^{-4}**	0.66
	34T	102.43	106.52	83.00	3.0×10^{-1}	0.54	**3.2×10^{-5}**	0.67	**1.0×10^{-4}**	0.66
	34F	102.43	106.52	83.00	3.0×10^{-1}	0.54	**3.2×10^{-5}**	0.67	**1.0×10^{-4}**	0.66
gimp_rgb_to_hsv_int	14T	22 459.23	460 789.00	431 509.54	**6.9×10^{-32}**	**0.02**	**1.1×10^{-28}**	**0.05**	7.0×10^{-1}	0.52
	17T	35.75	42.31	34.87	**1.4×10^{-3}**	*0.37*	2.0×10^{-1}	0.55	**2.0×10^{-5}**	0.67
gimp_hsv_to_rgb_int	4T	22.51	15.56	16.53	**1.7×10^{-13}**	**0.80**	**2.5×10^{-11}**	**0.77**	**9.6×10^{-3}**	*0.39*
	11T	20.31	81.31	24.93	**7.9×10^{-18}**	**0.15**	3.1×10^{-1}	0.46	**6.3×10^{-14}**	**0.81**
validate_card	5F	6.49	6.31	6.62	6.8×10^{-1}	0.52	8.7×10^{-1}	0.49	4.8×10^{-1}	0.47
	7T	5.59	5.99	5.89	4.7×10^{-1}	0.47	5.0×10^{-1}	0.47	9.4×10^{-1}	0.50
	7F	6.54	6.09	6.40	2.0×10^{-1}	0.55	6.9×10^{-1}	0.52	3.9×10^{-1}	0.47
	9T	6.38	6.12	6.59	5.3×10^{-1}	0.53	6.6×10^{-1}	0.48	2.7×10^{-1}	0.46
	9F	6.38	6.24	6.82	6.5×10^{-1}	0.52	3.1×10^{-1}	0.46	1.4×10^{-1}	*0.44*
	11T	6.50	6.46	6.46	1.0×10^{0}	0.50	9.4×10^{-1}	0.50	9.8×10^{-1}	0.50
	11F	5.92	6.04	6.24	7.8×10^{-1}	0.49	4.6×10^{-1}	0.47	6.0×10^{-1}	0.48
	14T	12.43	12.25	11.83	9.0×10^{-1}	0.51	3.4×10^{-1}	0.54	4.4×10^{-1}	0.53
	14F	7.30	6.55	6.81	8.0×10^{-2}	*0.57*	2.6×10^{-1}	0.55	5.0×10^{-1}	0.47

The largest improvements were observed with the *clip_to_circle*, with notable improvements in performance for *days_between* function.[4] This is remarkable since these branches contain the largest degree of multimodality (see Table 3). Here, it seems that the characteristics of the fitness function are generally less important than the range of input values. The above functions have large domain sizes for their input variables (as seen in Table 2), suggesting that we indeed see a difference in scalability between running times $\Theta((\log d)^2)$ and $O(\log d)$.

For the other functions, particularly *validates_card*, branches were covered relatively quickly regardless of the local search used, often leading to a difference of less than one evaluation on average—likely due to their function's small input range. Hence, the difference in runtime from a practical standpoint is almost negligible, and comparisons did not yield statistical significance.

Furthermore, there is one branch (namely branch 14T of the *gimp_rgb_to_hsv_int* test object) in which IPS performs far better than both Geometric and Lattice. The success rates for this branch are 0% for Random, 100% for IPS, 17% for Geometric, and 19% for Lattice. We observed that all searches frequently resorted to restarting. Further investigations with this branch and the Wegener Genetic Algorithm [9,31] revealed the GA was much more efficient at finding a solution, requiring only 4795 evaluations as a mean average, compared to 22 459 for the AVM with IPS. This result fits with those from previous studies in search-based test input generation, where the AVM works most efficiently for simple fitness landscapes with "obvious" optima, whereas diversifying GAs are more efficient at navigating less smooth landscapes generated by more difficult branches [9]. A detailed analysis of this branch is given in Section 7.

6.3. *Threats to the validity of the empirical study*

It is good practice in Software Engineering to discuss threats to validity associated with our empirical study, so that the reader may judge the limits to the claims that we make. From the point of view of *external threats*, the test objects in our experiments may not generalise in practice, however, care was taken to select them from a variety of real-world sources, from different programmers. These examples go beyond the bounds of our theory, but still show positive results

[4] Three branches of *days_between* share identical statistics: branches "34T" and "34F" correspond to the entry and exit point of a loop that is nested within branch "32T" and almost always subsequently entered.

in the majority of cases. From the point of view of *internal threats*, possible errors come from our implementation of the techniques. However, as shown with the simple and controlled `is_zero` example from Fig. 3(a), empirical results closely matched those expected from our theoretical observations. Furthermore we used non-parametric statistical tests to analyse our results, i.e., the Mann–Whitney U test and the Vargha–Delaney $\hat{A}_{12}$ statistic, both of which do not have assumptions regarding normality of the sample means, avoiding a further potential source of error in our analysis.

7. On the importance of finding the right local optimum

In our experiments branch 14T of the *gimp_rgb_to_hsv_int* test object was found to be challenging for the new AVM variants. We conduct a further analysis to find out why, leading to a rigorous explanation and further insights as to how choosing a different local optimum can impact on the global search behaviour.

In contrast to Sections 4 and 5, where the performance of a single local search run was studied, the performance analyses in this section concern the global search performance, where the goal is to cover the target branch. We focus on the number of starts AVM makes in order to cover the branch, as this is a major indicator that dominates search performance.

The branch 14T corresponds to the true condition of the following code (rewritten to ease presentation). It involves three variables r, g, b, which take integer values:

```
if (MAX(MAX(r, g), b) == 0) { /* TARGET BRANCH */ }
```

This means that the (non-normalised) branch distance for branch 14T equals the following minimisation problem:

$$\min\left\{\max(x_1, x_2, x_3) \mid x_1, x_2, x_3 \in [0, 255]\right\}$$

(we use $K = 0$ as K is irrelevant here). This function is hard to minimise by AVM: if there is a single variable with a maximum value among all coordinates, decreasing this variable decreases the maximum. However, decreasing any other variable does not improve the branch distance. Even worse, if there are several variables with the same maximal value, the branch distance cannot be improved by changing a single variable.

We formalise this in the following theorem, which considers a generalisation of the above settings to n variables and N values 0 to $N-1$. The theorem states that the success probability of AVM is exponentially small, provided that local search in this setting always returns the closest local optimum.

In our setting, we claim that both Geometric Search and Lattice Search will return the local optimum closest to the starting point. Let x_i be the variable AVM is currently optimising, and x_j be the largest variable different from x_i. Then local search is optimising $mf(x) = \max(x_i, x_j)$, and all $x_i \le x_j$ yield local optima. If local search starts with a local optimum, we return that optimum. Otherwise, $x_i > x_j$. The alternative local searches perform exploration starting with x_i towards decreasing values. When searching for a local minimum within an interval, ties are broken towards searching in the right part of the interval. This implies that the largest optimal value is returned.

Theorem 6. *Consider AVM minimising the function* $mf = \max(x_1, \dots, x_n)$ *with* $x_1, \dots, x_n \in [0, N-1]$. *Assume that the local search used in AVM outputs an optimum at a minimum distance to the starting point. Then the probability of AVM discovering the optimum* $(0, \dots, 0)$ *before restarting is* $(n \cdot N - n + 1) \cdot N^{-n}$. *Hence the expected number of starts is* $N^n/(n \cdot N - n + 1)$.

Proof. We claim that AVM finds the global optimum if and only if the initial solution has at least $n-1$ variables set to 0. Since the number of those vectors is $(n \cdot N - n + 1)$ and the probability of initialising with any specific vector is N^{-n}, this implies the claim.

It is easy to see that with this initial solution AVM manages to find the optimum. Assume there is one variable $x_i \neq 0$ in the initial search point. If AVM optimises a variable $x_j = 0$, local search stops at $x_j = 0$, and AVM moves on to the next variable. Once AVM starts a local search on x_i, since $\max(0, \dots, x_i, \dots, 0) = x_i$, local search will stop with $x_i = 0$.

Now assume AVM starts with a search point where the two largest variables are $x_i > x_j > 0$. Note that then $mf(x_1, \dots, x_n) = \max(x_i, x_j)$, and local search on any variable x_k, $k \neq i$, will stop and leave the current search point unchanged. Once AVM starts to optimise x_i, all values $x_i \le x_j$ lead to local optima of fitness x_j. Hence, by assumption AVM will stop with the closest local optimum, where $x_i = x_j$.

Once AVM has found a solution where the two largest values are equal and greater than 0, this solution cannot be improved by changing a single variable. Hence AVM will cycle through all variables and restart. □

So Theorem 6 implies the following for our setting of branch 14T.

Corollary 7. *The probability of AVM with either Geometric Search or Lattice Search solving branch 14T of gimp_rgb_to_hsv_int is* $(3 \cdot 256 - 3 + 1) \cdot 256^{-3} = 383/8\,388\,608$. *In expectation, around* 21 902 *starts are needed.*

The reason why AVM with IPS performs better on branch 14T is that IPS does not always stop at the largest local optimum. If the initial pattern search on a variable x_i finishes on a point x_i' on the plateau, IPS will stop at x_i'. This can

Table 5
Success probabilities of a run of AVM with IPS and the expected number of starts to find an optimum on $\max(x_1, x_2)$, where the range for both x_1 and x_2 is $[0, N-1]$ with $N = 2^1 - 1, 2^2 - 1, \ldots, 2^{16} - 1$. Figures are rounded.

Range	Success probability	Expected # starts
[0, 1]	0.75	1.33
[0, 3]	0.5625	1.78
[0, 7]	0.3906	2.56
[0, 15]	0.2617	3.82
[0, 31]	0.1729	5.79
[0, 63]	0.1135	8.81
[0, 127]	0.0744	13.44
[0, 255]	0.0487	20.52
[0, 511]	0.0319	31.35
[0, 1 023]	0.0209	47.90
[0, 2 047]	0.0137	73.18
[0, 4 095]	0.0089	111.81
[0, 8 191]	0.0059	170.83
[0, 16 383]	0.0038	261.01
[0, 32 767]	0.0025	398.78
[0, 65 535]	0.0016	609.29

leave AVM with a setting where then there is a unique maximum value x_j, in which case the next local search on x_j will decrease x_j, potentially towards a value $x'_j < x'_i$. Then x'_i might become the next unique maximum, and so forth. In other words, variables may go past one another, a phenomenon we call "leap-frogging".

For instance, when minimising the maximum of two variables in the 8-bit range, the input (166, 81), leading to the following trajectory:

$$(166, 81) \to (39, 81) \to (39, 18) \to (8, 18) \to (8, 3) \to (1, 3) \to (1, 0) \to (0, 0).$$

This trajectory can be further extended towards larger inputs by adding a value of $2^{j+1} - 1$ to the smaller input value, where $j \geq 1$ is chosen such that the difference between the two inputs is larger than 2^j.

Such long trajectories with leap-frogging are rare as in every step AVM needs to avoid setting two variables to the same value. But it is also clear that because of leap-frogging AVM with IPS has a strictly higher success probability than AVM with Geometric Search or Lattice Search. The same effect also occurs with more than two variables, however the chances of setting two variables to the same value increase with the number of variables.

We investigated the case of two variables in more detail and determined success probabilities for AVM with IPS numerically for ranges $[0, N-1]$ with $N = 2^1, 2^2, \ldots, 2^{16}$, by running AVM on all possible inputs. The results are shown in Table 5, including the expected number of starts for AVM to be successful.

Theorem 6 gives a success probability $2/N - N^{-2}$ for AVM with Geometric Search and Lattice Search, hence an expected number of starts of around $N/2$. Compared to this value, the times in Table 5 are much lower for AVM with IPS.

To conclude this section, the alternative local searches still speed up the time for finding *some* local optimum. But branch 14T turns out to be an example where for the global search behaviour *which* local optimum is found makes a big difference. Both Geometric Search and Lattice Search find the worst possible local optimum in a sense that AVM gets stuck on points with two or more variables sharing the maximum value. AVM with IPS may perform better as variables can perform leap-frogging and avoid such search points. In any case, minimising the maximum of n variables is a problem not well suited for optimisation methods that only change one coordinate at a time (as previously noted in the empirical evaluation).

8. Conclusions and future work

We have analysed the performance of the original AVM incorporating Iterated Pattern Search (IPS), proposing to replace the latter with faster local searches, Geometric and Lattice Search. On strictly unimodal functions, these searches provably need less time than IPS. On every strictly unimodal function, IPS requires time $\Theta((\log d)^2)$ in the worst case, when the initial point can be chosen up to a distance of d. The same holds for its average-case performance on the easy unimodal function $f(x) = |x|$. In contrast to this, the alternative searches succeed in time $O(\log d)$ on any strictly unimodal function, where d is the initial distance to the optimum. These theoretical results closely matched the results of experiments optimising the easy function $f(x) = |x|$ and variants thereof.

We further empirically analysed AVM with Geometric Search and Lattice Search on test objects that gave rise to unimodal as well as multimodal functions. For multimodal functions there are no non-trivial performance guarantees for any local search; our experiments therefore extend the realm of what can be proven theoretically. Considering branches where any variant of AVM performed significantly better than random search, we found that Geometric and Lattice performed significantly better than IPS on a majority of branches. Results varied with the input domain size, which determines the initial distance to global optima. For small domain sizes of only 10 values, no statistically significant differences were found.

But for larger domains, significant differences emerged: AVM with Lattice Search clearly outperformed AVM with IPS on all branches with input domain size at least 2^{16}. Excluding the pathological case of branch 14T in *gimp_rgb_to_hsv_int*, AVM with Lattice Search needed less than 50% evaluations overall, compared to the original AVM. The reason why IPS performed better on branch 14T was found to be that the alternative local searches returned different local optima than IPS, which on this specific branch hampered global search performance. The branch was found to be challenging for all AVM variants as it contains large plateaux, which AVM cannot escape from by changing a single variable.

One idea for future research is to use the alternative local searches for the AVM to further improve results with Memetic Algorithms (MAs) [8,9], which combine diversifying GA searches with intensifying local search algorithms [24,28]. Such an approach was found to provide the "best of both worlds" for test input generation in Harman and McMinn's study [9], and it may help to deal with plateaux as above. Thus, further work is needed to investigate the performance of the alternative local searches with the AVM when integrated into an MA. Another idea is to investigate different, possibly randomised tie-breaking rules to eliminate worst-case scenarios as experienced with branch 14T.

References

[1] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008.
[2] A. Arcuri, Full theoretical runtime analysis of alternating variable method on the triangle classification problem, in: International Symposium on Search Based Software Engineering, SSBSE, 2009, pp. 113–121.
[3] A. Arcuri, P.K. Lehre, X. Yao, Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem, in: IEEE International Workshop on Search-Based Software Testing, SBST, 2008, pp. 161–169.
[4] A. Auger, B. Doerr (Eds.), Theory of Randomized Search Heuristics – Foundations and Recent Developments, Series on Theoretical Computer Science, vol. 1, World Scientific, 2011.
[5] M. Dietzfelbinger, J. Rowe, I. Wegener, P. Woelfel, Precision, local search and unimodal functions, Algorithmica 59 (2011) 301–322.
[6] M. Dietzfelbinger, J.E. Rowe, I. Wegener, P. Woelfel, Tight bounds for blind search on the integers and the reals, Combin. Probab. Comput. 19 (2010) 711–728.
[7] D.E. Ferguson, Fibonaccian searching, Commun. ACM 3 (12) (1960) 648.
[8] G. Fraser, A. Arcuri, P. McMinn, A memetic algorithm for whole test suite generation, J. Syst. Softw. (2015), http://dx.doi.org/10.1016/j.jss.2014.05.032, in press.
[9] M. Harman, P. McMinn, A theoretical and empirical study of search based testing: local, global and hybrid search, IEEE Trans. Softw. Eng. 36 (2) (2010) 226–247.
[10] T. Jansen, Analyzing Evolutionary Algorithms – The Computer Science Perspective, Springer, 2013.
[11] J. Kempka, P. McMinn, D. Sudholt, A theoretical runtime and empirical analysis of different alternating variable searches for search-based testing, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '13, ACM, 2013, pp. 1445–1452.
[12] J. Kiefer, Sequential minimax search for a maximum, Proc. Amer. Math. Soc. 4 (1953) 502–506.
[13] B. Korel, Automated software test data generation, IEEE Trans. Softw. Eng. 16 (8) (1990) 870–879.
[14] P.K. Lehre, X. Yao, Crossover can be constructive when computing unique input-output sequences, Soft Comput. 15 (2011) 1675–1687.
[15] P.K. Lehre, X. Yao, Runtime analysis of the $(1+1)$ EA on computing unique input output sequences, Inform. Sci. 259 (2014) 510–531.
[16] G. McGraw, C. Michael, M. Schatz, Generating software test data by evolution, IEEE Trans. Softw. Eng. 27 (12) (2001) 1085–1110.
[17] P. McMinn, Search-based software test data generation: a survey, Softw. Testing Verification Reliab. 14 (2) (2004).
[18] P. McMinn, IGUANA: input generation using automated novel algorithms. A plug and play research tool, Technical report CS-07-14, Department of Computer Science, University of Sheffield, 2007, http://www.dcs.shef.ac.uk/intranet/research/public/resmes/CS0714.pdf.
[19] P. McMinn, An identification of program factors that impact crossover performance in evolutionary test input generation for the branch coverage of C programs, Inf. Softw. Technol. 55 (1) (2013).
[20] P. McMinn, M. Stevenson, M. Harman, Reducing qualitative human oracle costs associated with automatically generated test data, in: International Workshop on Software Test Output Validation, STOV 2010, ACM, 13 July 2010, pp. 1–4.
[21] L.L. Minku, D. Sudholt, X. Yao, Evolutionary algorithms for the project scheduling problem: runtime analysis and improved design, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '12, ACM, 2012, pp. 1221–1228.
[22] L.L. Minku, D. Sudholt, X. Yao, Improved evolutionary algorithm design for the project scheduling problem based on runtime analysis, IEEE Trans. Softw. Eng. 40 (1) (2014) 83–102.
[23] J.F. Monahan, Numerical Methods of Statistics, 2 edition, Cambridge University Press, 2011.
[24] F. Neri, C. Cotta, P. Moscato (Eds.), Handbook of Memetic Algorithms, Studies in Computational Intelligence, vol. 379, Springer, 2012.
[25] F. Neumann, C. Witt, Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity, Springer, 2010.
[26] R. Pargas, M. Harrold, R. Peck, Test-data generation using genetic algorithms, Softw. Testing Verification Reliab. 9 (4) (1999) 263–282.
[27] S. Steinarsson, Downsampling time series for visual representation, M. Sc. thesis, University of Iceland, 2013, http://hdl.handle.net/1946/15343.
[28] D. Sudholt, Memetic evolutionary algorithms, in: A. Auger, B. Doerr (Eds.), Theory of Randomized Search Heuristics – Foundations and Recent Developments, in: Series on Theoretical Computer Science, vol. 1, World Scientific, 2011.
[29] N. Tracey, J. Clark, K. Mander, The way forward for unifying dynamic test case generation: the optimisation-based approach, in: International Workshop on Dependable Computing and Its Applications, 1998, pp. 169–180.
[30] A. Vargha, H.D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong, J. Educ. Behav. Stat. 25 (2) (2000) 101–132.
[31] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Inf. Softw. Technol. 43 (14) (2001) 841–854.