

# Full Theoretical Runtime Analysis of Alternating Variable Method on the Triangle Classification Problem

Andrea Arcuri

School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.

email: a.arcuri@cs.bham.ac.uk

## Abstract

Runtime Analysis is a type of theoretical investigation that aims to determine, via rigorous mathematical proofs, the time a search algorithm needs to find an optimal solution. This type of investigation is useful to understand why a search algorithm could be successful, and it gives insight of how search algorithms work. In previous work, we proved the runtimes of different search algorithms on the test data generation for the Triangle Classification (TC) problem. We theoretically proved that Alternating Variable Method (AVM) has the best performance on the coverage of the most difficult branch in our empirical study. In this paper, we prove that the runtime of AVM on all the branches of TC is  $O((\log n)^2)$ . That is necessary and sufficient to prove that AVM has a better runtime on TC compared to the other search algorithms we previously analysed. The theorems in this paper are useful for future analyses. In fact, to state that a search algorithm has worse runtime compared to AVM, it will be just sufficient to prove that its lower bound is higher than  $\Omega((\log n)^2)$  on the coverage of at least one branch of TC.

## 1 Introduction

Although there has been a lot of research on *Search Based Software Engineering* [6, 2, 5] in recent years (e.g. in software testing [15]), there exists few theoretical results. The only exceptions we are aware of are on computing unique input/output sequences for finite state machines [13, 12], the application of the Royal Road theory to evolutionary testing [7], and our previous work on test data generation for the Triangle Classification (TC) problem [1].

To get a deeper understanding of the potential and limitations of the application of search algorithms in software engineering, it is essential to complement the existing experimental research with theoretical investigations. *Runtime Analysis* is an important part of this theoretical investiga-

tion, and brings the evaluation of search algorithms closer to how algorithms are classically evaluated.

The goal of analysing the runtime of a search algorithm on a problem is to determine, via rigorous mathematical proofs, the *time* the algorithm needs to find an optimal solution. In general, the runtime depends on characteristics of the problem instance, in particular the problem instance *size*. Hence, the outcome of runtime analysis is usually expressions showing how the runtime depends on the instance size. This will be made more precise in the next sections.

The field of runtime analysis has now advanced to a point where the runtime of relatively complex search algorithms can be analysed on classical combinatorial optimisation problems [20]. We advocate that this type of analysis in Search Based Software Engineering will be helpful to get insight on how search algorithms behave in the software engineering domain. The final aim is to exploit the gained knowledge to design more efficient algorithms.

Branch coverage is the testing task we want to solve. We do a different search for each branch in the code. The employed fitness function is the commonly used approximation level plus the branch distance [15]. Because the number of branches is a constant, the overall runtime for the fulfilment of the test criterion is given by the most expensive search. The size of the problem is given by the constraints on the range of the input variables.

In our previous work [1], we proved runtimes for three different search algorithms on the coverage of one branch of the TC problem. The analysed search algorithms are: Random Search, Hill Climbing and Alternating Variable Method (AVM). The analysed branch is the one related to the classification of the triangle as equilateral (that empirically seems the most difficult to cover).

In that previous work, we proved that AVM has a runtime of  $O((\log n)^2)$ , that is strictly better than the ones of the other search algorithms we analysed (i.e.,  $\Theta(n)$  for Hill Climbing and  $\Theta(n^2)$  for Random Search). However, that is not sufficient to claim that AVM has a better runtime on TC. In fact, although it has been empirically shown that the “equilateral branch” is the most difficult to cover, that is not

necessarily true for high values of the size that have not been empirically tested. Other branches might be more difficult. This is one reason why theoretical analyses are necessary.

In this paper, we prove that the expected runtime of AVM on all the branches of TC is  $O((\log n)^2)$ . That is necessary and sufficient to prove that AVM has a better runtime on TC compared to the other search algorithms we previously analysed. We also carried out an empirical study to integrate the theoretical analysis.

The theorems in this paper are also useful for future analyses. In fact, to state that a search algorithm has worse runtime compared to AVM, it will be just sufficient to prove that its lower bound is higher than  $\Omega((\log n)^2)$  on the coverage of at least one branch of TC.

The paper is organised as follows. Section 2 gives background information about runtime analysis. Section 3 describes in detail the TC problem, whereas Section 4 describes the AVM search algorithm. Theoretical analyses are presented in Section 5, whereas the empirical study is discussed in Section 6. Finally, Section 7 concludes the paper.

## 2 Runtime Analysis

To make the notion of runtime precise, it is necessary to define time and size. We defer the discussion on how to define problem instance size for software testing to the next section, and define time first.

Time can be measured as the number of basic operations in the search heuristic. Usually, the most time-consuming operation in an iteration of a search algorithm is the evaluation of the cost function. We therefore adopt the *black-box scenario* [4], in which time is measured as the number of times the algorithm evaluates the cost function.

**Definition 1** (Runtime [3, 8]). *Given a class  $\mathcal{F}$  of cost functions  $f_i : S_i \rightarrow \mathbb{R}$ , the runtime  $T_{A,\mathcal{F}}(n)$  of a search algorithm  $A$  is defined as*

$$T_{A,\mathcal{F}}(n) := \max \{T_{A,f} \mid f \in \mathcal{F} \text{ with } \ell(f) = n\},$$

where  $\ell(f)$  is the problem instance size, and  $T_{A,f}$  is the number of times algorithm  $A$  evaluates the cost function  $f$  until the optimal value of  $f$  is evaluated for the first time.

A typical search algorithm  $A$  is randomised. Hence, the corresponding runtime  $T_{A,\mathcal{F}}(n)$  will be a random variable. The runtime analysis will therefore seek to estimate properties of the distribution of random variable  $T_{A,\mathcal{F}}(n)$ , in particular the *expected runtime*  $E[T_{A,\mathcal{F}}(n)]$  and the *success probability*  $\Pr[T_{A,\mathcal{F}}(n) \leq t(n)]$  for a given time bound  $t(n)$ . More details can be found in [1].

The last decades of research in the area show that it is important to apply appropriate mathematical techniques to get good results [22]. Initial studies of exact Markov chain

models of search heuristics were not fruitful, except for the simplest cases.

A more successful and particularly versatile technique has been so-called drift analysis [8, 19], where one introduces a potential function which measures the distance from any search point to the global optimum. By estimating the expected one-step drift towards the optimum with respect to the potential function, one can deduce expected runtime and success probability.

In addition to drift analysis, the wide range of techniques used in the study of randomised algorithms [17], in particular Chernoff bounds, have proved useful also for evolutionary algorithms.

## 3 Triangle Classification Problem

TC is the most famous problem in software testing. It opens the classic 1979 book of Myers [18], and has been used and studied since early 70s. Nowadays, TC is still widely used in many publications (e.g., [14, 23, 15, 11, 16, 24]).

We use the implementation for the TC problem that was published in the survey by McMinn [15] (see Figure 1). Some slight modifications to the program have been introduced for clarity.

A solution to the testing problem is represented as a vector  $I = (x, y, z)$  of three integer variables. We call  $(a, b, c)$  the permutation in ascending order of  $I$ . For example, if  $I = (3, 5, 1)$ , then  $(a, b, c) = (1, 3, 5)$ .

There is the problem to define what is the *size* of an instance for TC. In fact, the goal of runtime analysis is not about calculating the exact number of steps required for finding a solution. On the other hand, the runtime complexity of an algorithm gives us insight of scalability of the search algorithm. The problem is that TC takes as input a fixed number of variables, and the structure of its source code does not change. Hence, what is the *size* in TC? We chose to consider the range for the input variables for the size of TC. In fact, it is a common practise in software testing to put constraints on the values of the input variables to reduce the search effort. For example, if a function takes as input 32 bit integers, instead of doing a search through over four billion values, a range like  $\{0, \dots, 1000\}$  might be considered for speeding up the search.

Limits on the input variables are always present in the form of bit representation size. For example, the same piece of code might be either run on machine that has 8 bit integers or on another that uses 32 bits. What will happen if we want to do a search for test data on the same code that runs on a 64 bit machine? Therefore, using the range of the input variables as the size of the problem seems an appropriate choice.

In our analyses, the size  $n$  of the problem defines the range  $R = \{-n/2 + 1, \dots, n/2\}$  in which the variables in  $I$  can be chosen (i.e.,  $x, y, z \in R$ ). Hence, the search space  $S$  is defined as  $S = \{(x, y, z) | x, y, z \in R\}$ , and it is composed of  $n^3$  elements. Without loss of generality  $n$  is even. To obtain full coverage, it is necessary that  $n \geq 8$ , otherwise the branch regarding the classification as *scalene* will never be covered. Note that one can consider different types of  $R$  (e.g.,  $R' = \{0, \dots, n\}$ ), and each type may lead to different behaviours of the search algorithms. We based our choice on what is commonly used in literature. For simplicity and without loss of generality, search algorithms are allowed to generate solutions outside  $S$ . In fact,  $R$  is mainly used when random solutions need to be initialised.

The employed fitness function  $f$  is the commonly used approximation level  $\mathcal{A}$  plus the branch distance  $\delta$  [15]. For a target branch  $z$ , we have that the fitness function  $f_z$  is:

$$f_z(I) = \mathcal{A}_z(I) + \omega(\delta_w(I)).$$

Note that the branch distance  $\delta$  is calculated on the node of *diversion*, i.e. the last node in which a critical decision (not taking the branch  $w$ ) is made that makes the execution of  $z$  not possible. For example, branch  $z$  could be nested to a node  $N$  (in the control flow graph) in which branch  $w$  represents the *then* branch. If the execution flow reaches  $N$  but then the *else* branch is taken, then  $N$  is the node of diversion for  $z$ . The search hence gets guided by  $\delta_w$  to enter in the nested branches.

Let be  $\{N_0, \dots, N_k\}$  the sequence of diversion nodes for the target  $z$ , with  $N_i$  nested to all  $N_{j>i}$ . Let be  $D_i$  the set of inputs for which the computation diverges at node  $N_i$  and none of the nested nodes  $N_{j<i}$  is executed. Then, it is important that  $\mathcal{A}_z(I_i) < \mathcal{A}_z(I_j) \forall I_i \in D_i, I_j \in D_j, i < j$ . A simple way to guarantee it is to have  $\mathcal{A}_z(I_{i+1}) = \mathcal{A}_z(I_i) + \zeta$ , where  $\zeta$  can be any positive constant (e.g.,  $\zeta = 1$ ) and  $\mathcal{A}_z(I_0) = 0$ .

Because an input that makes the execution closer to  $z$  should be rewarded, then it is important that  $f_z(I_i) < f_z(I_{i+1}) \forall I_i \in D_i, I_{i+1} \in D_{i+1}$ . To guarantee that, we need to scale the branch distance  $\delta$  with a scaling function  $\omega$  such that  $0 \leq \omega(\delta_j) < \zeta$  for any predicate  $j$ . Note that  $\delta$  is never negative. We need to guarantee that the order of the values does not change once mapped with  $\omega$ , for example  $h_0 > h_1$  should imply  $\omega(h_0) > \omega(h_1)$ . We can use for example either  $\omega(h) = (\zeta h)/(h+1)$  or  $\omega(h) = \zeta/(1+e^{-h})$ , where  $h \geq 0$ .

Having  $\zeta > 0$  and  $\gamma > 0$ , the fitness functions for the 12 branches (i.e.,  $f_i$  is the fitness function for branch  $ID_i$ ) are shown in Figure 2. Note that the branch distance depends on the status of the computation (e.g., the values of the local variables) when the predicates are evaluated. For simplicity, in an equivalent way we show the fitness functions based only on the inputs  $I$ .

```

1: int tri_type(int x, int y, int z) {
2:   int type;
3:   int a=x, b=y, c=z;
4:   if (x > y) { /* ID_0 */
5:     int t = a; a = b; b = t;
6:   } else { /* ID_1 */}
7:   if (a > z) { /* ID_2 */
8:     int t = a; a = c; c = t;
9:   } else { /* ID_3 */}
10:  if (b > c) { /* ID_4 */
11:    int t = b; b = c; c = t;
12:  } else { /* ID_5 */}
13:  if (a + b <= c) { /* ID_6 */
14:    type = NOT_A_TRIANGLE;
15:  } else { /* ID_7 */
16:    type = SCALENE;
17:    if (a == b && b == c) {
18:      /* ID_8 */
19:      type = EQUILATERAL;
20:    } else /* ID_9 */
21:      if (a == b || b == c) {
22:        /* ID_10 */
23:        type = ISOSCELES;
24:      } else { /* ID_11 */}
25:    }
26:  return type;
27: }
```

**Figure 1. Triangle Classification (TC) program, adapted from [15]. Each branch is tagged with a unique ID.**

---


$$f_0(I) = \begin{cases} 0 & \text{if } x > y, \\ \omega(|y - x| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_1(I) = \begin{cases} 0 & \text{if } x \leq y, \\ \omega(|x - y| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_2(I) = \begin{cases} 0 & \text{if } \min(x, y) > z, \\ \omega(|z - \min(x, y)| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_3(I) = \begin{cases} 0 & \text{if } \min(x, y) \leq z, \\ \omega(|\min(x, y) - z| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_4(I) = \begin{cases} 0 & \text{if } \max(x, y) > \max(z, (\min(x, y))), \\ \omega(|\max(z, (\min(x, y))) - \max(x, y)| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_5(I) = \begin{cases} 0 & \text{if } \max(x, y) \leq \max(z, (\min(x, y))), \\ \omega(|\max(x, y) - \max(z, (\min(x, y)))| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_6(I) = \begin{cases} 0 & \text{if } a + b \leq c, \\ \omega(|(a + b) - c| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_7(I) = \begin{cases} 0 & \text{if } a + b > c, \\ \omega(|c - (a + b)| + \gamma) & \text{otherwise.} \end{cases}$$

$$f_8(I) = \begin{cases} \zeta + f_7(I) & \text{if } a + b \leq c, \\ 0 & \text{if } a == b \wedge b == c \wedge a + b > c, \\ \omega(|a - b| + |b - c| + 2\gamma) & \text{otherwise.} \end{cases}$$

$$f_9(I) = \begin{cases} \zeta + f_7(I) & \text{if } a + b \leq c, \\ 0 & \text{if } (a \neq b \vee b \neq c) \wedge a + b > c, \\ \omega(2\gamma) & \text{otherwise.} \end{cases}$$

$$f_{10}(I) = \begin{cases} 2\zeta + f_7(I) & \text{if } a + b \leq c, \\ \zeta + f_9(I) & \text{if } a == b \wedge b == c \wedge a + b > c, \\ 0 & \text{if } (a \neq b \vee b \neq c) \wedge a + b > c \wedge (a == b \vee b == c), \\ \omega(\min(|a - b| + \gamma, |b - c| + \gamma)) & \text{otherwise.} \end{cases}$$

$$f_{11}(I) = \begin{cases} 2\zeta + f_7(I) & \text{if } a + b \leq c, \\ \zeta + f_9(I) & \text{if } a == b \wedge b == c \wedge a + b > c, \\ 0 & \text{if } a \neq b \wedge b \neq c \wedge a + b > c, \\ \omega(\gamma) & \text{otherwise.} \end{cases}$$


---

**Figure 2. Fitness functions  $f_i$  for all the branches  $ID_i$  of TC. The constants  $\zeta$  and  $\gamma$  are both positive, and  $0 \leq \omega(h) < \zeta$  for any  $h$ .**

## 4 Alternating Variable Method

AVM is similar to a Hill Climbing, and was employed in the early work of Korel [10]. The algorithm starts on a random search point  $I$ , and then it considers modifications of the input variables, one at a time. The algorithm applies an *exploratory search* to the chosen variable, in which the variable is slightly modified (in our case, by  $\pm 1$ ). If one of the neighbours has a better fitness, then the exploratory search is considered successful. Similarly to a Hill Climbing, the better neighbour will be selected as the new current solution. Moreover, a *pattern search* will take place. On the other hand, if none of the neighbours has better fitness, then AVM continues to do exploratory searches on the other variables, until either a better neighbour has been found or all the variables have been unsuccessfully explored. In the latter case, a restart from a new random point is done if a global optimum was not found.

A pattern search consists of applying increasingly larger changes to the chosen variable as long as a better solution is found. The type of change depends on the exploratory search, which gives a direction of growth. For example, if a better solution is found by decreasing the input variable by 1, then the following pattern search will focus on decreasing the value of that input variable.

A pattern search ends when it does not find a better solution. In this case, AVM will start a new exploratory search on the same input variable. In fact, the algorithm moves to consider one other variable only in the case that an exploratory search is unsuccessful.

To simplify the writing of the AVM implementation, and for making it more readable, it is not presented in its general form. Instead, it is specialised in working on vector solutions of length three. The general version, that considers this length as a problem parameter, would have the same computational behaviour in terms of evaluated solutions.

**Definition 2** (Alternating Variable Method (AVM)).

```

while termination criterion not met
  Choose  $I$  uniformly in  $S$ .
  while  $I$  improved in last 3 loops
     $i :=$  current loop index.
    Choose  $T_i \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$  such that
       $T_i \neq T_{i-1} \wedge T_i \neq T_{i-2}$ .
     $found := true$ .
    while  $found$ 
      for  $d := 1$  and  $d := -1$ 
         $found := exploratory\_search(T_i, d, I)$ .
      if  $found$ , then
         $pattern\_search(T_i, d, I)$ 

```

**Definition 3** (*exploratory\_search*( $T_i, d, I$ )).

```

 $I' := I + dT_i$ .
if  $f(I') \geq f(I)$ , then
  return false.
else
   $I := I'$ .
  return true.

```

**Definition 4** (*pattern\_search*( $T_i, d, I$ )).

```

 $k := 2$ .
 $I' := I + kdT_i$ .
while  $f(I') < f(I)$ 
   $I := I'$ .
   $k := 2k$ .
   $I' := I + kdT_i$ .

```

## 5 Theoretical Analysis

**Proposition 1.** *Given  $x$  and  $y$  uniformly and independently distributed in  $R$ , then their expected difference with  $y \geq x$  is  $E[y - x | y \geq x] = \frac{n-1}{3} = \Theta(n)$ . The largest difference would be  $y - x = n - 1 = \Theta(n)$*

*Proof.*

$$\begin{aligned}
 E[y - x | y \geq x] &= (n + 2(n-1) + 3(n-2) + \dots + n(1)) \\
 &\quad / \frac{n(n+1)}{2} - 1 \\
 &= \sum_{i=0}^n (i+1)(n-i) \cdot \frac{2}{n(n+1)} - 1 \\
 &= \frac{n(n+1)(n+2)}{6} \cdot \frac{2}{n(n+1)} - 1 \\
 &= \frac{n-1}{3} \\
 &= \Theta(n).
 \end{aligned}$$

The highest value that  $y$  can take is  $n/2$ . The lowest value  $x$  can take is  $-n/2 + 1$ . Hence,  $n/2 - (-n/2 + 1) = n - 1$ .  $\square$

**Lemma 1.** *For any branch, if the probability that the random starting point is a global optimum is lower bounded by a positive constant  $k > 0$ , then AVM needs at most a constant number  $\Theta(1)$  of restarts to find a global optimum.*

*Proof.* If we consider only the starting point, the AVM behaves as a random search, in which the probability of finding a global optimum is bigger than  $k$ . That can be described as a Bernoulli process (see our theorems on random search in [1]), with expected number of restarts that is lower or equal than  $1/k$ .  $\square$

**Theorem 1.** *The expected time for AVM to find an optimal solution to the coverage of branches  $ID\_0$  and  $ID\_1$  is  $O(\log n)$ .*

*Proof.* The probability that  $x > y$ , with  $X$  and  $Y$  the random variables representing them, is:

$$\begin{aligned} \Pr[X > Y] &= \sum_y \sum_{i=y+1}^{n/2} \Pr[X = i \mid Y = y] \cdot \Pr[Y = y] \\ &= \frac{1}{2} - \frac{1}{2n}, \end{aligned}$$

The probability of  $x \leq y$  is hence  $\frac{1}{2} + \frac{1}{2n}$ .

Considering that the search is done for values of  $n$  bigger or equal than 8, then both the searches for  $ID_0$  and  $ID_1$  start with a random point that is a global optimum with a probability lower bounded by a positive constant. Therefore, AVM needs at most a constant number of restarts (Lemma 1), independently of the presence and number of local optima.

For both branches, either the starting point is a global optimum, or the search will be influenced by the distance  $x - y$  that is  $\Theta(n)$  (Proposition 1). We hence analyse this latter case.

Until the predicate is not satisfied, the fitness function  $\omega(|y - x| + \gamma)$  (with  $\gamma$  a positive constant) based on the branch distance rewards any reduction of that distance. The third variable  $z$  does not influence the fitness function, hence an exploratory search fails on that. For the coverage of  $ID_0$ , the variable  $x$  has gradient to increase its value, and  $y$  has gradient to decrease. For  $ID_1$  it is the opposite. The distance  $|x - y|$  can be covered in  $O(\log n)$  steps of a pattern search.  $\square$

**Theorem 2.** *The expected time for AVM to find an optimal solution to the coverage of branches  $ID_2$ ,  $ID_3$ ,  $ID_4$  and  $ID_5$  is  $O(\log n)$ .*

*Proof.* The sentences in lines 5, 8 and 11 of the source code (Figure 1) only swap the value of the three input variables. Hence, the predicate conditions of branches  $ID_2$ ,  $ID_3$ ,  $ID_4$  and  $ID_5$  are directly based on the values of two different input variables.

The type of predicates is the same of branches  $ID_0$  and  $ID_1$  (i.e.,  $>$ ), and the condition of the comparison is the same (i.e.,  $>$  on two input variables). The three input variables are uniformly and independently distributed in  $R$ , and by Proposition 1 the maximum distance among them is  $\Theta(n)$ . There are the same conditions of Theorem 1 apart from the fact that the variables could be swapped during the search, i.e. the fact that lines 5 and 8 are executed or not can vary during the search.

For branches  $ID_2$  and  $ID_3$ , starting from  $z$  no variation of the executed code is done until the branch is covered. For branch  $ID_3$ , for either  $x$  or  $y$  a search starting from the maximum of them would result in no improvement

of the fitness function. The minimum of  $x$  and  $y$  has a gradient to decrease, and while it does so the relation of their order is not changed. Hence, no variation of the executed code is done. On the other hand, for branch  $ID_2$ , the minimum has gradient to increase, but the pattern search would stop once it becomes the maximum of the two (e.g.,  $x > y$  if the search started on  $x$  with  $x < y$ ). That happens in at most  $O(\log n)$  steps because their difference is at most  $\Theta(n)$  (Proposition 1). If the next variable considered by AVM is not  $z$ , then the above behaviour will happen again. However, the next variable will be necessarily  $z$ , hence we have at most  $O(\log n)$  steps done 3 times, that still results in  $O(\log n)$  steps.

For any pair of values we have that  $\min(x, y) \leq \max(x, y)$ . For branch  $ID_4$ , if it is not executed, then  $\max(x, y) \leq \max(z, \min(x, y))$  and necessarily it would be  $z \geq \max(x, y) \geq \min(x, y)$ . Hence,  $z$  would have gradient to decrease down until  $\max(x, y)$ , in which case  $ID_4$  gets executed after  $O(\log n)$  steps. A modification of the minimum value between  $x$  and  $y$  does not change the fitness value. For the maximum value, it can increase up to  $z$ , in which case  $ID_4$  gets executed after  $O(\log n)$  steps. The relation of the order of the input variables would not be changed during those searches.

For branch  $ID_5$ , if it is not executed, then  $\max(x, y) > \max(z, \min(x, y))$  and necessarily it would be  $x \neq y$  and  $\max(x, y) > z$ . Starting the search from the maximum of  $x$  and  $y$  would have gradient to decrease down to  $\max(z, \min(x, y))$ , that would be done in  $O(\log n)$  steps that will make  $ID_5$  executed. If  $z < \min(x, y)$ , modifying  $z$  would have no effect to the fitness function, whereas the minimum of  $x$  and  $y$  has gradient to increase up to  $\max(x, y)$ . In the other case  $z \geq \min(x, y)$ , it is the other way round, i.e.  $z$  can increase whereas the minimum between  $x$  and  $y$  cannot change. In both cases, in  $O(\log n)$  steps branch  $ID_5$  gets executed with no change in the relation of the order of the input variables.

The expected time for branches  $ID_2$ ,  $ID_3$ ,  $ID_4$  and  $ID_5$  is therefore the same as for branches  $ID_0$  and  $ID_1$ , i.e.  $O(\log n)$ .  $\square$

**Theorem 3.** *The expected time for AVM to find an optimal solution to the coverage of branch  $ID_6$  is  $O(\log n)$ .*

*Proof.* If the predicate  $a + b \leq c$  is not true, the fitness function would be  $\omega(|a + b - c| + \gamma)$  (with  $\gamma$  a positive constant). For values  $a \leq 0$ , the predicate is true because  $a + b \leq b \leq c$ .

There is gradient to decrease  $a$  and  $b$ , and there is gradient to increase  $c$ . If the search starts from either  $a$  or  $b$ , in  $O(\log n)$  steps of a pattern search the target variable assumes a negative value (the highest possible starting value is  $n/2$ ). In particular, if the search starts from  $b$ , at a certain

point the input variable representing  $b$  will instead represent  $a$ . Otherwise, it sufficient to increase  $c$  up to the value  $a + b \leq n/2 + n/2 = n$ , that can be done in  $O(\log n)$  steps of a pattern search.  $\square$

**Theorem 4.** *The expected time for AVM to find an optimal solution to the coverage of branch  $ID_8$  is  $O((\log n)^2)$ .*

*Proof.* This theorem has been proved in our previous work [1].  $\square$

**Lemma 2.** *For search algorithms that use the fitness function only for direct comparisons of candidate solutions, the expected time for covering a branch  $ID_w$  is not higher than the expected time to cover any of its nested branches  $ID_z$ .*

*Proof.* Before a target nested branch  $ID_z$  is executed, its “parent” branch  $ID_w$  needs to be executed. Until  $ID_w$  is not executed, the fitness function  $f_z^w$  (i.e., search for  $ID_z$  and  $ID_w$  is not covered) will be based on the predicate of the branch  $ID_w$ . Hence, that fitness function would be equivalent to the one  $f_w$  used for a direct search for  $ID_w$ . In particular,  $f_z^w(I) = \zeta + f_w(I)$ , that because the approximation level would be different. However, because the constant  $\zeta > 0$  would be the same to all the search points, the behaviour of a search algorithm, that uses the fitness function only for direct comparisons of candidate solutions, would be same on these two fitness functions (and AVM satisfies this constraint).

Because the time to solve (i.e., finding an input that minimises)  $f_z^w$  is not higher than the time needed for  $f_z$  and because  $f_z^w$  is equivalent to  $f_w$ , then solving  $f_w$  cannot take in average more time than solving  $f_z$ .  $\square$

**Theorem 5.** *The expected time for AVM to find an optimal solution to the coverage of branch  $ID_7$  is  $O((\log n)^2)$ .*

*Proof.* The branch  $ID_8$  is nested to branch  $ID_7$ , hence by Lemma 2 and Theorem 4 the expected time is  $O((\log n)^2)$ .  $\square$

**Theorem 6.** *The expected time for AVM to find an optimal solution to the coverage of branch  $ID_9$  is  $O((\log n)^2)$ .*

*Proof.* By Theorem 5, the branch  $ID_7$  can be covered in  $O((\log n)^2)$  steps. The branch  $ID_9$  (that is nested to  $ID_7$ ), will be covered if  $\neg(a = b \wedge b = c)$ . If that predicate is not true, a single exploratory search of AVM makes it true because it is just sufficient to either increase or decrease any input variable by 1. The only case in which this is not possible is for  $I = (1, 1, 1)$ , because it is the only solution that satisfies  $a = b \wedge b = c \wedge a - 1 + b \leq c \wedge a + b \leq c + 1 \wedge a + b > c$ . In that case, a restart is done.

With a probability that is lower bounded by a constant, in a random starting point each input variable is higher than  $n/4$ . In that case,  $a + b > c$ , because  $(n/4) + 1 + (n/4) + 1 > (n/2)$ . By Lemma 1, we need only  $\Theta(1)$  restarts.  $\square$

**Theorem 7.** *The expected time for AVM to find an optimal solution to the coverage of branch  $ID_{10}$  is  $O((\log n)^2)$ .*

*Proof.* By Theorem 6, the branch  $ID_9$  can be covered in  $O((\log n)^2)$  steps. The branch  $ID_{10}$  (that is nested to  $ID_9$ ), will be covered if only two input variables are equal (and not all three equal to each other at the same time).

If when the branch  $ID_7$  (branch  $ID_9$  is nested to it) is executed all the three input variables are equal (in that case branch  $ID_8$  is executed), then a single exploratory search is sufficient to execute branch  $ID_{10}$ , because we just need to change the value of a single variable.

The other case in which all the three variables are different is quite complex to analyse. Instead of analysing it directly, we prove the runtime by a comparison with the behaviour of AVM on the branch  $ID_8$  (that is more complex and we already proved it in our previous work [1]).

Once branch  $ID_9$  is executed, the fitness function  $f_{10}^9$  for covering branch  $ID_{10}$  is based on  $\min(\delta(a = b), \delta(b = c))$ , whit  $\delta$  the branch distance function for the predicates. For simplicity, let consider  $\delta(a = b) < \delta(b = c)$ . The other case can be studied in the same way.

An exploratory search cannot accept a reduction of the distance  $c - b$ , because the value of  $f_{10}^9$  would not improve. A search on  $a$  would leave the distance  $c - b$  unchanged. About  $b$ , only a decrease of its value would be accepted, and in that case the distance  $c - b$  would increase (but that has no effect on the fitness function because it takes the minimum of the two distances). Because the branch distance  $\delta$  only rewards the reduction of the distance  $b - a$ , a search starting from either  $a$  or  $b$  will end in  $a = b$  by modifying only the value of only one of these variables (AVM keeps doing searches on the same variable till an exploratory search fails). During that search, the fitness function would hence be based on  $\delta(a = b)$ .

In a search for covering branch  $ID_8$ , if the branch  $ID_7$  (in which both  $ID_8$  and  $ID_{10}$  are nested) is executed, then the fitness function  $f_8^7$  depends on  $\delta(a = b) + \delta(b = c)$ . A search starting from  $a$  would finish in  $a = b$  for the same reasons explained before or it would finish in  $a' > b$  (with  $a'$  the latest accepted point for  $a$  that will become the new  $b$  in the next exploratory search). During that search, the value of  $\delta(b = c)$  does not change, so it can be considered as a constant. Because AVM uses the fitness function only on direct comparisons, the presence of a constant does not influence its behaviour. Therefore, in this particular context (i.e.,  $\delta(a = b) < \delta(b = c)$ ), branch  $ID_7$  executed and

search starting from  $a$ ) the behaviour of AVM on  $f_{10}^9$  and  $f_8^7$  will be the same until  $a = b$  or  $a' > b$ .

In the case  $a = b$ , branch  $ID_{10}$  gets executed and the search for that branch ends. In the other case  $a' > b$ , the previous  $b$  becomes the new  $a_k$  and  $a'$  becomes the new  $b_k$ . Modifications on the variable  $c$  does not change the value of either  $a_k$  or  $b_k$ . The previous analysis can hence be recursively applied to the new values  $a_k$  and  $b_k$ . If  $a' > c$ , then  $a_k = b$ ,  $b_k = c$ ,  $c_k = a'$  and it will become the case  $\delta(a = b) > \delta(b = c)$ .

It is still necessary to analyse the behaviour of AVM on  $f_{10}^9$  when the search starts on  $b$  rather than  $a$ . That is similar to the case of  $f_8^7$  when the variable  $c$  is decreased down to  $b$ . In that context, the two fitness functions are of the same type because in  $f_8^7$  the distance  $\delta(a = b)$  would be a constant until  $b = c$  or  $c' < b$  (with  $c'$  the latest accepted point for  $c$  that will become the new  $b$  in the next exploratory search). Therefore, the runtime for AVM on  $f_{10}^9$  to obtain  $a = b$  would be the same.

By Theorem 4, the expected time for covering branch  $ID_{10}$  is  $O((\log n)^2)$ . Because we proved that the coverage of  $ID_{10}$  takes more time than the coverage of  $ID_{11}$ , then the expected runtime for covering  $ID_{11}$  is  $O((\log n)^2)$ .  $\square$

**Theorem 8.** *The expected time for AVM to find an optimal solution to the coverage of branch  $ID_{11}$  is  $O((\log n)^2)$ .*

*Proof.* By Theorem 6, the branch  $ID_9$  can be covered in  $O((\log n)^2)$  steps. The branch  $ID_{11}$  (that is nested to  $ID_9$ ), will be covered if  $a \neq b \wedge a \neq c \wedge b \neq c$ . In the moment that the branch  $ID_9$  is executed, then the three variables cannot assume all the same value (otherwise the branch  $ID_8$  would have been executed). If that predicate is not true, a single exploratory search of AVM makes it true because it is just sufficient to increase by 1 any of the two variables that have same values.  $\square$

## 6 Empirical Study

We ran an implementation of AVM on each branch of TC for values of  $n$  such that  $n = 2^i$ , with  $i \in \{4, 5, 6, \dots, 29, 30\}$ . For each size of  $n$ , we ran 1000 trials (with different random seeds) and recorded the number of fitness evaluations done before reaching a global optimum.

Following [9], for each setting of algorithm and problem instance size, we fitted different models to the observed runtimes using non-linear regression with the Gauss-Newton algorithm. Each model corresponds to a one term expression  $\eta \cdot g(n)$  of the runtime, where the model parameter  $\eta$  corresponds to the constant to be estimated. The residual sum of squares of each fitted model was calculated to identify the model which corresponds best with the observed

**Table 1. Results of empirical experiments.**

Branch ID	Runtime
ID_0	$0.48 \log_2 n$
ID_1	$0.50 \log_2 n$
ID_2	$1.03 \log_2 n$
ID_3	$0.36 \log_2 n$
ID_4	$0.34 \log_2 n$
ID_5	$1.62 \log_2 n$
ID_6	$0.10 \log_2 n$
ID_7	$5.31 \log_2 n$
ID_8	$0.53(\log_2 n)^2$
ID_9	$5.10 \log_2 n$
ID_10	$7.47 \log_2 n$
ID_11	$5.01 \log_2 n$

runtimes. This methodology was implemented in the statistical tool  $R$  [21].

The used models were  $\eta n^t \log(n)^v$ , where  $t \in \{0, 1, 2\}$  and  $v \in \{0, \dots, 10\}$ . The models with lowest error are shown in Table 6.

The results in Table 6 are consistent with our theoretical results. They are able to provide the constants for the runtime models. It is worth noting that running so many experiments (i.e., 324,000) was possible because the runtime of AVM is  $O((\log n)^2)$ . In the case for example of a runtime  $\Theta(n^2)$  (e.g., Random Search [1]), running so many experiments would have likely been unfeasible. The resulting estimated models could have been hence not precise.

At any rate, we ran our experiments only with values of  $n$  up to  $2^{30}$ . We cannot know for sure what could happen for higher values. On the other hand, our theoretical analysis is valid for each value of  $n$ .

## 7 Conclusions and Future Work

In this paper, we proved that the runtime of AVM on all the branches of TC is  $O((\log n)^2)$ . This is necessary and sufficient to prove that AVM has a better runtime on TC compared to the other search algorithms we previously analysed, i.e. Random Search and Hill Climbing. In the future, to state that a search algorithm performs worse than AVM on TC, it will be just sufficient to prove that its lower bound is higher than  $\Omega((\log n)^2)$  on the coverage of at least one branch of TC.

Previously, we proved that AVM requires on average  $O((\log n)^2)$  steps for covering the branch related to the classification of the triangle as equilateral ( $ID_8$ ). However, we needed to prove the runtime on each single branch, because  $ID_8$  is not necessarily the most difficult to cover. Although empirical studies in literature have shown that

branch *ID.8* seems the most difficult to cover, that is not sufficient, because different behaviours might arise for very high values of the size.

This type of analysis is important to understand the behaviour of search algorithms on software engineering problems. Unfortunately, the fact that they are difficult to carry out limits its scope. Therefore, theoretical runtime analysis is not meant to replace empirical studies. However, for the problems for which theoretical analyses can be done, we get stronger and more reliable results than any obtained with empirical studies.

For the future, we want to analyse other search algorithms, in particular Genetic Algorithms. Considering other implementation of TC would be interesting to see if there is any difference in the overall runtimes of the analysed algorithms. In the long term, it will be interesting to analyse more complex software to get more insight on how search algorithms work.

## 8 Acknowledgements

The author is grateful to Xin Yao and Per Kristian Lehre for insightful discussions. This work is supported by EP-SRC grant EP/D052785/1.

## References

- [1] A. Arcuri, P. K. Lehre, and X. Yao. Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem. In *International Workshop on Search-Based Software Testing (SBST)*, pages 161–169, 2008.
- [2] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.
- [3] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276:51–81, 2002.
- [4] S. Droste, T. Jansen, and I. Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory of Computing Systems*, 39(4):525–544, 2006.
- [5] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pages 342–357, 2007.
- [6] M. Harman and B. F. Jones. Search-based software engineering. *Journal of Information & Software Technology*, 43(14):833–839, 2001.
- [7] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 73–83, 2007.
- [8] J. He and X. Yao. A study of drift analysis for estimating computation time of evolutionary algorithms. *Natural Computing*, 3(1):21–35, 2004.
- [9] T. Jansen. On the brittleness of evolutionary algorithms. In *Foundations of Genetic Algorithms*, pages 54–69, 2007.
- [10] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.
- [11] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from b formal models: Research articles. *Software Testing, Verification and Reliability*, 14(2):81–103, 2004.
- [12] P. Lehre and X. Yao. Crossover can be constructive when computing unique input output sequences. In *Proceedings of the International Conference on Simulated Evolution and Learning (SEAL)*, pages 595–604, 2008.
- [13] P. K. Lehre and X. Yao. Runtime analysis of (1+1) ea on computing unique input output sequences. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1882–1889, 2007.
- [14] J. C. Lin and P. L. Yeh. Automatic test data generation for path testing using GAs. *Information Sciences*, 131(1-4):47–64, 2001.
- [15] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.
- [16] J. Miller, M. Reformat, and H. Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, 48(7):586–605, 2006.
- [17] M. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [18] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.
- [19] P. Oliveto and C. Witt. Simplified drift analysis for proving lower bounds in evolutionary computation. In *Proceedings of the international conference on Parallel Problem Solving from Nature*, pages 82–91, 2008.
- [20] P. S. Oliveto, J. He, and X. Yao. Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results. *International Journal of Automation and Computing*, 4(3):281–293, 2007.
- [21] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.
- [22] I. Wegener. Methods for the analysis of evolutionary algorithms on pseudo-boolean functions. Technical Report CI-99/00, Universität Dortmund, 2000.
- [23] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.
- [24] M. Xiao, M. El-Attar, M. Reformat, and J. Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.