

# Theoretical Runtime Analyses of Search Algorithms on the Test Data Generation for the Triangle Classification Problem

Andrea Arcuri, Per Kristian Lehre and Xin Yao

The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA),  
School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.  
email: {a.arcuri,p.k.lehre,x.yao}@cs.bham.ac.uk

## Abstract

*Software Testing plays an important role in the life cycle of software development. Because software testing is very costly and tedious, many techniques have been proposed to automate it. One technique that has achieved good results is the use of Search Algorithms. Because most previous work on search algorithms has been of an empirical nature, there is a need for theoretical results that confirm the feasibility of search algorithms applied to software testing. Such theoretical results might shed light on the limitations and benefits of search algorithms applied in this context. In this paper, we formally analyse the expected runtime of three different search algorithms on the problem of Test Data Generation for an instance of the Triangle Classification program. The search algorithms that we analyse are Random Search, Hill Climbing and Alternating Variable Method. We believe that this is a necessary first step that will lead and help the Software Engineering community to better understand the role of Search Based Techniques applied to software testing.*

## 1 Introduction

Software Testing is used to find the presence of bugs in computer programs [16]. If no bug is found, testing cannot guarantee that the software is bug-free. However, testing can be used to increase our confidence in the software reliability. Unfortunately, testing is expensive, time consuming and tedious. It is estimated that testing requires around 50% of the total cost of software development [1]. This cost is paid because testing is very important. Releasing bug-ridden and non-functional software is indeed an easy way to lose customers. For example, in the USA alone it is estimated that every year around \$20 billion could be saved if better testing was done before releasing new software [19].

White Box Testing [16] is a type of testing in which the quality of a test suite is based on structural criteria. One

common criterion is *branch coverage*, in which we look for a test suite that when run executes each branch in the code of the tested program. The reason for doing white box testing is that bugs can lie in parts of code that are rarely executed. Hence, they are difficult to spot during the development cycle. Our analyses in this paper focus only on branch coverage.

Many techniques to automate the testing phase have been proposed. Among them, modelling the test problem as a search problem has got widespread consideration in the last few years [2]. In fact, search algorithms such as Genetic Algorithms (GAs) [9] have been successfully applied to solve many complex tasks in several different engineering domains, hence it is reasonable to apply them on Software Engineering problems. Although their application in software testing has given promising results so far [14], most of the research work on that subject has been of empirical nature. The only exceptions we are aware of are on computing unique input/output sequences for finite state machines [12], and on the application of the Royal Road theory to evolutionary testing [7].

In this paper we theoretically analyse the runtime behaviour of some search algorithms applied to an implementation of the Triangle Classification (TC) problem [16]. We chose TC because it is the most famous problem in software testing. The search algorithms considered for the analyses are: Random Search (RS), Hill Climbing (HC) and Alternating Variable Method (AVM).

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.

To get a deeper understanding of the potential and limitations of the application of search algorithms in software engineering, it is necessary to complement the existing ex-

perimental research with theoretical investigations. Runtime analysis is important part of this theoretical investigation, and brings the evaluation of search algorithms closer to how algorithms are classically evaluated. During the last decade, there has been much research on runtime analysis of randomised search algorithms. The field has now advanced to a point where the runtime of relatively complex search algorithms can be analysed on classical combinatorial optimisation problems [17].

In search based white box testing, in the case of branch coverage, it is common to tackle each different branch separately. In other words, there will be a different search for each different branch. However, analyses on the dependencies graph can be used to choose only a sub-set of branches. In fact, the execution of a particular branch might imply the execution of others. In such a case, a successful search for covering that branch necessarily implies the coverage of others, hence they do not need separated searches. Because there is a constant number of branches, the runtime of a search algorithm is determined only by the most expensive search.

The main contributions of this paper are:

- To our best knowledge this is the first work on runtime analyses of search algorithms applied to Software Test Data Generation.
- We formally proved that exists at least one search algorithm (e.g., HC and AVM) that has a runtime complexity that is strictly better than the one of RS on at least one important test problem (i.e, TC).

The paper is organised as follows. Section 2 gives background about runtime analysis. Section 3 describes in detail the TC problem, whereas section 4 describes and analyses the use of three different search algorithms applied to find test data for TC. Finally, section 5 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 Section 3, 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* [5], in which time is measured as the number of times the algorithm evaluates the cost function.

**Definition 1** (Runtime [4, 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, i.e. the runtime depends on the random bits used. 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  $t(n)$ .

## 3 Triangle Classification Problem

TC is the most famous problem in software testing. It opens the classic 1979 book of Myers [16], and has been used/studied since at least the early 70s (e.g., [6, 18, 3]). However, the true origin of TC is not completely clear [21]. At any rate, nowadays TC is still widely used in many publications (e.g., [13, 20, 14, 11, 15, 22]), although that is more likely due to the lack of organisation of the community in preparing a proper benchmark suite.

The used implementation for the TC problem is the one published in survey by McMinn [14] (see figure 1).

In the case of the analysed implementation of TC, the most difficult branch to cover is the one related to the classification of the triangle as *equilateral*. Although this is commonly believed to be so, it might not be necessarily be the hardest branch to cover. Moreover, what is difficult for a particular search algorithm might be very easy for others. Hence, the difficulty of a test problem cannot be analysed without considering the applied search algorithms. However, for the rest of the paper we will consider that hypothesis as true for the search algorithms analysed in this paper. Therefore, we can limit our studies only on the coverage of that branch, because from that we can infer the overall complexity of the analysed search algorithms.

A solution to the 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 practice 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 the over four billions values, a range like  $\{0, \dots, 1000\}$  might be considered for speeding up the search.

Although limits on the input variables are common in software testing, usually there is no guarantee that there exists a global optimum within those limits. However, how fast the runtime of a search algorithm increases, when the range of the variables is increased, gives us useful information. For example, what are the consequences of choosing a too wide range?

At any rate, 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 bit. 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  $S = \{(x, y, z) | x, y, z \in R\}$ . To obtain full coverage, it is necessary that  $n \geq 8$ , otherwise the branch regarding the classification as *scalene* will never be covered. To note that different types of  $R$  could be considered (e.g.,  $R' = \{0, \dots, n\}$ ), and for each type there would be different behaviours of the search algorithms. We based our choice on what is commonly done in literature.

The search space is composed by  $n^3$  elements. However, instead of considering  $n$ , we could use  $q$  with  $2^{q-1} < n \leq 2^q$ , where  $q$  represents the max number of bits allowed for the input variables. In that case, the search space would be large  $2^{3q}$ . In our analyses, we prefer to consider  $n$  instead of  $q$  because we think it is more clear.

As defined in [14], the objective function used here is the sum of the approach level with the normalised branch distance. In our particular case, the objective function  $f$  to minimise is:

$$f(V) = \begin{cases} 1 + \omega(d(\neg(a + b \leq c))) & \text{if } a + b \leq c, \\ 0 + \omega(d(a = b \vee b = c)) & \text{otherwise,} \end{cases} \quad (1)$$

where:

$$d(\neg(a + b \leq c)) = \begin{cases} 0 & \text{if } c - a - b < 0, \\ (c - a - b) + K & \text{otherwise,} \end{cases} \quad (2)$$

$$d(a = b \wedge b = c) = \begin{cases} 0 & \text{if } a = b, \\ \text{abs}(a - b) + K & \text{otherwise,} \\ + \begin{cases} 0 & \text{if } b = c, \\ \text{abs}(b - c) + K & \text{otherwise,} \end{cases} \end{cases}$$

```
int tri_type(int a, int b, int c) {
    int type;
    if (a > b) { int t = a; a = b; b = t; }
    if (a > c) { int t = a; a = c; c = t; }
    if (b > c) { int t = b; b = c; c = t; }
    if (a + b <= c) {
        type = NOT_A_TRIANGLE;
    } else {
        type = SCALENE;
        if (a == b && b == c) {
            type = EQUILATERAL;
        } else if (a == b || b == c) {
            type = ISOSCELES;
        }
    }
    return type;
}
```

**Figure 1. Triangle Classification Program [14].**

$$\omega(h) = \frac{1}{1 + e^{-h}}. \quad (3)$$

$K$  can be any arbitrary positive constant (e.g.,  $K = 1$ ), and the input  $h$  can be any real number. Note that, for Eq. (3), any normalising function in the range  $[0, 1]$  can be used. Moreover, if a search algorithm uses the fitness values only for direct comparisons (as is the case for all the search algorithms described in this paper), the choice of the normalising function does not have any effect besides its computational cost. An example, for which this would not apply, is the use a “Fitness Proportional Selection” in GAs.

## 4 Search Algorithms

There are many search algorithms, and for each algorithm there are several different variants.

To simplify the writing of the search algorithm implementations, and for making them more readable, they are not presented in their general form. Instead, they are specialised in working on vector solutions of length three. However, the general versions, that consider that length as a problem parameter, would have the same computational behaviour in term of evaluated solutions.

The runtime of the algorithm is defined as the number of iterations until the optimum has been found for the first time. Hence, the choice of the termination criterion does not influence the runtime, and is therefore left unspecified to simplify the description of the algorithms.

**Theorem 1.** *Given the objective function (1) and the space of solutions  $R$ , there are  $n/2$  global optima, and they are on the form  $G = (t, t, t)$ , with  $t > 0$ .*

*Proof.* This can be proved by considering fitness function (1), in which the minimal fitness value is given for  $(a + b > c) \wedge (a = b) \wedge (b = c)$ . The points  $G$  are the only that satisfy  $(a = b) \wedge (b = c)$ , and  $t + t > t$  implies  $t > 0$ . Because the range of the variables is  $R = \{-n/2 + 1, \dots, n/2\}$ , there are  $n/2$  possible different  $t$  with  $t > 0$ .  $\square$

The two following simple properties of the problem will be used extensively in the runtime analysis.

**Proposition 1.** *Let  $a \leq b \leq c$ , and  $v > 0$ , then  $f(a, b, c) < f(a - v, b, c)$ .*

*Proof.* In the case when  $a + b \leq c$ , then  $a - v + b \leq c$  and we have  $f(a, b, c) = 1 + \omega(c - a - b + K)$  and  $f(a - v, b, c) = 1 + \omega(c - (a - v) - b + K)$ , in which case the proposition holds. Assume on the other hand that  $a + b > c$ . Let  $g$  be:

$$g = \begin{cases} 0 & \text{if } a = b \wedge b = c, \\ 2K & \text{if } a \neq b \wedge b \neq c, \\ K & \text{otherwise,} \end{cases}$$

we get:

$$\begin{aligned} f(a - v, b, c) &\geq \omega(c - a + v + g) \\ &> \omega(c - a + g) \\ &= f(a, b, c). \end{aligned}$$

$\square$

**Proposition 2.** *If  $a \leq b \leq c$ , and  $v > 0$ , then  $f(a, b, c) < f(a, b, c + v)$ .*

*Proof.* In the case when  $a + b \leq c$ , we have:

$$\begin{aligned} f(a, b, c + v) &= 1 + \omega(v + c - a - b + K) \\ &> 1 + \omega(c - a - b + K) \\ &= f(a, b, c). \end{aligned}$$

For the opposite case  $a + b > c$ , we have:

$$\begin{aligned} f(a, b, c + v) &\geq \omega(v + c - a + g) \\ &> \omega(c - a + g) \\ &= f(a, b, c), \end{aligned}$$

where  $g$  is as defined in the proof of Proposition 1.  $\square$

## 4.1 Random Search

RS is the easiest search algorithm. It simply samples search points at random, and stops when a global optimum is found (i.e., when the target branch is covered). RS does not exploit any information got so far by the visited points when choosing the next to sample. Often, RS is used as a

baseline for evaluating the performance of other more sophisticated meta-heuristics.

At any rate, it is important to not confuse RS in white box testing with Random Testing (RT). In RT, in fact, random points (i.e., test cases) are sampled, and those will compose the final test suite. On the other hand, in our case we use RS to find and choose test cases for getting the highest possible branch coverage.

**Definition 2** (Random Search (RS)).

**while** *termination criterion not met*  
Choose  $I$  uniformly from  $S$ .

**Theorem 2.** *The expected time for RS to find an optimal solution to TC is  $\Theta(n^2)$ .*

*Proof.* The probability of getting three identical values is  $1/n^2$ . The probability that a point  $I = (a, a, a)$  has  $a > 0$  is  $1/2$ . Therefore, the probability that a random search point is a global optimum is  $1/2n^2$ .

The behaviour of RS can therefore be described as a Bernoulli process, where the probability of getting a global optimum for the first time after  $t$  steps is geometrically distributed  $\Pr [T_{RS} = t] = (1 - 1/2n^2)^{t-1} \cdot (1/2n^2)$ . Hence, the expected time for RS to find a global optimum is  $E[T_{RS}] = 2n^2$ .  $\square$

**Theorem 3.** *The probability that RS has found an optimal solution to TC within  $n^3$  iterations is exponentially large  $1 - e^{-\Omega(n)}$ .*

*Proof.* Using the inequality  $(1 - 1/x)^x \leq e^{-1}$ , it is easy to see that  $\Pr [T_{RS} > n^3] = \left(1 - \frac{1}{2n^2}\right)^{n^3} \leq e^{-n/2}$ .  $\square$

## 4.2 Hill Climbing

HC is a search algorithm that belongs to the class of *local search* algorithms. That means that given a starting point  $I_0$ , it looks at neighbour solutions  $N(I_0)$  that are “near” to  $I_0$ . If a better solution  $I' \in N(I_0)$  exists, then the next point  $I_1$  will be that  $I'$ . Then, the same procedure of looking at the neighbour solutions is done on  $I_1$ , until a final point  $I_i$  is reached, in which  $\forall I' \in N(I_i) : f(I') \geq f(I_i)$ , assuming we want to minimise function  $f$ . This means that no neighbour solution is better, and the algorithm is said to be stuck in either a local or global optimum. If  $I_i$  is not a global optimum, then HC can restart from a new different point  $I_0$ .

At any rate, HC is not a single specific algorithm, but a family of algorithms. In fact, we need to define how the neighbourhood  $N$  is generated, the strategy  $\delta$  for visiting  $N$ , and finally how to do the restarts.

Given that the solution is a vector of integers (of length three in our particular case), an appropriate neighbourhood is  $N(I_i) := \{I_i + d \mid d \in D \text{ and } I_i + d \in S\}$ , where  $D := \{(\pm 1, 0, 0), (0, \pm 1, 0), (0, 0, \pm 1)\}$ .

A random restart is a common choice, and we use it for the HC that we analyse. Regarding the strategy  $\delta$ , we do not need to define one. In fact, the following analyses of HC are valid for all strategies satisfying the following constraint: unless a new better solution is found, each neighbour solution will be visited in at most a constant number of iterations (assuming that the neighbourhood size is constant). The implication is very straightforward: if the current point  $I_i$  is neither a local or global optimum, then a better solution will be found in at most a constant number of iterations. Note that this constraint is very common, and most of the HC variants satisfy it.

**Definition 3** (Hill Climbing (HC)).

**while** *termination criterion not met*  
*Choose I uniformly at random from S.*  
**while** *I not a local optimum in N(I),*  
*Choose I' from N(I) according to strategy  $\delta$*   
**if**  $f(I') < f(I)$ , **then**  
 $I := I'$ .

**Theorem 4.** *The expected time for HC to find an optimal solution to TC is  $\Theta(n)$ .*

*Proof.* We first need to prove that all the points of the form  $L = (t, t, t)$  with  $t \leq 0$  are local optima. Because for all of them  $a + b \leq c$  is true, we have  $f(L) = 1 + \omega(-t + K)$ . Any operation on the vector  $I$  can either increase  $c$  by one, or decrease  $a$  by one. In both the cases, the resulting points  $L'$  have worse fitness (Proposition 1 and 2), that is  $f(L') = 1 + \omega(-t + 1 + K)$ . Because  $f(L') > f(L)$ , the points  $L$  are local optima.

Considering Propositions 1 and 2, a solution  $I'$  is not accepted if the value of  $a$  has decreased, or if the value of  $c$  has increased. Moreover, there is always a gradient for  $a$  to increase up to  $b$ , and for  $c$  to decrease down to  $b$ , because there would be a fitness improvement whatever  $a + b \leq c$  is true or not. Although the value of  $b$  can either increase or decrease, its number of changes is finite, because  $a \leq b \leq c$  is always true and because HC accepts as new solutions only strictly better points. Therefore, after a finite number of steps (i.e., the algorithm does not enter in an infinite loop, like it would happen if new solutions with equal fitness would be accepted), the current solution  $I$  converges to a point of the form  $W = (t, t, t)$ , with  $a \leq t \leq c$ . If  $b$  does not change during the search, then  $t = b$ .

Although we already proved that all the points in the form  $(t, t, t)$  are either local or global optima, only after the discussion in the previous paragraph we can state that  $L$  are the only possible local optima. In fact, regardless of the

starting point, HC reaches either  $L$  or  $G$ , and  $G$  are global optima.

A step is called successful if the new search point  $I'$  is accepted. The number of successful steps  $\eta$  for HC to reach an optimum depends on how the value of  $b$  changes. If it does not change, then there are  $b - a$  steps in which  $a$  increases, and  $c - b$  steps in which  $c$  decreases. Hence,  $\eta = c - a$ . There is only one case in which  $b$  can decrease:  $(a + b > c) \wedge (b = a + 1)$ , because in all other cases the fitness would never be better. If  $b$  is decreased before  $a$  increases (that depends on how the strategy  $\delta$  works), then  $\eta = 1 + c - a$ , because then  $a = b$  and  $a$  and  $b$  cannot be changed again. If it is still  $(a + b > c)$  but  $(b \neq a + 1)$ , then  $b$  cannot be altered until  $a$  is increased up to  $b - 1$ , because the fitness would not change. On the other hand, while  $(a + b \leq c)$ , there is always a gradient for  $b$  to increase up to  $c - a + 1$ . However, if  $a \leq 0$ ,  $b$  can increase up to  $c$ . Again, depending on  $\delta$ , it is possible that  $c$  decreases before  $b$  increases, and vice-versa. In the worst case with  $a = b$  and  $a \leq 0$ , we can have  $\eta = 2(c - a)$ , because  $b$  can take  $c - a$  steps to increase up to  $c$ , and other  $c - a$  steps for  $a$  to increase up to  $c$  as well. Therefore, regardless of the starting point  $I$  and strategy  $\delta$ , the number  $\eta$  of successful steps is bounded by  $(c - a) \leq \eta \leq 2(c - a)$ .

Unless the algorithm is stuck in an optimum, in at most a constant number of iterations it will find a better solution in its neighbourhood. Considering the bounds of  $\eta$ , the expected number of iterations for reaching an optimum is  $\Theta(c - a)$ . This difference can be expressed in terms of problem size  $n$  by noting that the expected value of  $a$  is  $E[a \mid b] = b/2 - (n/2 + 1)/2$ , whereas  $E[c \mid b] = b + (n/2 - b)/2$ . It now follows that  $E[c - a \mid b] = E[c \mid b] - E[a \mid b] = (n + 1)/2$ , independently of  $b$ . Therefore, starting from a random point, the expected number of iterations for reaching either a local or a global optimum is  $\Theta(n)$ .

When an optimum is reached, HC does a restart if that point is a local optimum. Therefore, we need to calculate the number of restarts that are required for HC to find an optimal solution.

If  $c \leq 0$ , then HC is bound to reach a local optimum regardless of the strategy  $\delta$ . That happens because it will reach a point  $(t, t, t)$  with  $t \leq c$ . Because  $c \leq 0$  implies  $t \leq 0$ , then that point is a local optimum. With the same type of reasoning, if  $a > 0$ , then HC is bound to find a global optimum. We said that there is only one case in which  $b$  can decrease up to  $a$ , and that is  $b = a + 1$ . However, because for doing it there is the need of  $a + b > c$ , then  $a > 0$  is required. Therefore, if  $b > 0$ , then  $b$  will always remain a positive value. Hence, we can generalise the condition of reaching a global optimum from  $a > 0$  to a more significant  $b > 0$ . Note that  $a > 0$  implies  $b > 0$ , but the opposite is not always true.

There is still to consider the case  $(b \leq 0) \wedge (c > 0)$ , in which the result is actually depending on the strategy  $\delta$ . If it chooses to decrease  $c$  at least down to 0 before increasing  $b$  up to 1, then a local optimum will be reached, or a global optimum if it chooses to do the opposite. However, as we will show, the analysis of that situation is not important for finding a lower and an upper bound for the number of required restarts.

The probability of starting from a point with  $c \leq 0$  is  $P(c \leq 0) = \frac{1}{8}$ . On the other hand, the probability of starting from a point with  $b > 0$  is equivalent at the probability of flipping a coin three times and getting at least two heads, hence,  $P(b > 0) = \frac{1}{8} + 3\frac{1}{8} = \frac{1}{2}$ . Therefore, regardless of the strategy  $\delta$ , we have that the probability of reaching a global optimum from a random point is  $\frac{1}{2} \leq P(\text{global}) \leq \frac{3}{4}$ , whereas for reaching a local optimum it is  $\frac{1}{8} \leq P(\text{local}) \leq \frac{1}{2}$ .

Therefore, the expected number of restarts is no more than 2. Because reaching either a local or global optimum from a random point requires  $\Theta(n)$  steps, and the expected number of restarts to reach a global optimum is no more than 2, it follows that the expected runtime of HC is on TC  $\Theta(n)$ .  $\square$

**Theorem 5.** *The probability that HC has found an optimal solution to TC within  $c \cdot n^2$  iterations is exponentially large  $1 - e^{-\Omega(n)}$ , where  $c$  is a constant.*

*Proof.* The time to reach a local optimum is at most  $c \cdot n$  iterations, where the constant  $c$  is determined by the strategy  $\delta$ . The probability that HC finds a local optimum more than  $n$  times before a global optimum is found is less than  $2^{-n} = e^{-\Omega(n)}$ .  $\square$

### 4.3 Alternating Variable Method

AVM is similar to an HC, and was employed in the early work of Korel [10]. The algorithm starts on a random search point  $I$ , and then it considers the modifications of the input variables one at the time. It applies to the chosen variable an *exploratory search*, in which that variable is slightly modified (in our case, by  $\pm 1$ ). If one of the neighbours has a better fitness, then the exploratory search is successful. As it happens in HC, that better neighbour will be selected as the new current solution. Moreover, a *pattern search* will take place. Otherwise, AVM continues to do exploratory searches on the other variables, until either a better neighbour has been found or all the variable have been unsuccessfully explored. In that latter case, a restart from a new random point is done if a global optimum was not found.

A pattern search consists of applying larger changes to that variable and increase the size of the changes after each new 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 applying a  $-1$  to the input variable, 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 that 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.

**Definition 4** (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 5** ( $exploratory\_search(T_i, d, I)$ ).

```

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

```

**Definition 6** ( $pattern\_search(T_i, d, I)$ ).

```

 $k := 2$ .
 $I' := I + kdT_i$ .
while  $I' \in S \wedge f(I') < f(I)$ 
   $I := I'$ .
   $k := 2k$ .
   $I' := I + kdT_i$ .

```

**Theorem 6.** *The expected time of AVM for finding an optimal solution is  $\Omega(\log n)$  and  $O((\log n)^2)$ .*

*Proof.* We first start to prove that, before doing a restart, AVM converges to a solution in the form  $T = (t, t, t)$ , where  $a \leq t \leq c$ . The discussion is similar to the proof done for HC. The variable representing  $a$  can only increase (Proposition 1), and has a gradient to increase up to  $b$ , i.e., each succession of increments of  $a$  has better fitness (that can be easily proved with an induction on Proposition 1). Similarly,  $c$  cannot increase (Proposition 2), and it has a

gradient to decrease down to  $b$ , and although  $b$  can change,  $b$  will still be in the interval  $[a, c]$ .

The difference is that, during the search, the input variables might take values lower than the starting  $a$  and bigger than  $c$ . For example, in the latter case that might happen if the variable  $x$  (for example) representing  $b$  gets an high increase, and that increase will make  $x$  bigger than  $c$ . However, in that case  $x$  (now representing the new  $c'$ ) will have a gradient to be decreased at least down to the original  $c$  (that now has become the new  $b'$ ). Therefore, even if a variable can get a value bigger than  $c$ , it will be immediately decreased afterwards. That happens because AVM works on the same variable till its change can improve the fitness. The case of having values lower than the initial  $a$  can be discussed in the same way.

Because AVM accepts only strictly better solutions, there will be a finite number of steps before converging to  $T$ . Moreover, once a random (re-)starting point is chosen, the behaviour of AVM is deterministic. Therefore, the probability of finding either a local or global optimum depends only on the starting point.

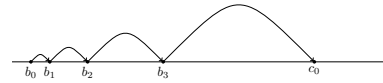
Following the same discussion done for HC, both these probabilities are lower and upper bounded by constants. In particular, if we just consider the cases  $a > 0$  and  $c \leq 0$ , we have  $\frac{1}{8} \leq P(\text{global}) \leq \frac{3}{4}$ . Hence, in expectation, AVM needs a constant number  $\Theta(1)$  of restarts to reach a global optimum in the same way as HC.

For discussing the convergence to  $T$ , we consider two opposite cases in which only one of the following opposite predicates holds:  $a + b \leq c$  and  $a + b > c$ . We separately study the runtime of each of these cases. The reason for doing that is to simplify the proof. In fact, during the search it can happen only once that from a solution that satisfies  $a + b \leq c$  we move to a case in which  $a + b > c$  is true, and the vice-versa is not possible. That happens because, due to the fact that  $\omega(x) < 1$  for all  $x$ , all the solutions satisfying the latter predicate have a strictly better fitness value than the cases in which  $a + b \leq c$  hold.

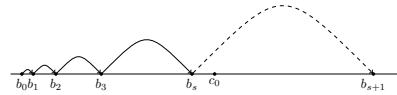
Although we can separately study the expected runtime of these two cases, what we are actually interested in is the overall runtime. We will prove that in both cases we get the same runtime  $\Omega(\log n)$  and  $O((\log n)^2)$ . Hence, because there might be only one swap from  $a + b \leq c$  to  $a + b > c$ , the overall asymptotic runtime will be the same.

In the case of  $a + b \leq c$ , we will consider the change of each variable separately. As a reminder, we have three input variables  $(x, y, z)$ , which represent the ordered values  $(a, b, c)$ . AVM works on the inputs  $(x, y, z)$ , and if for example  $x$  represents the lowest value  $a$  and it gets increased, then it might represent  $b$  in successive steps of the search. With  $(a_0, b_0, c_0)$  we consider the ordered values of  $(x, y, z)$  when AVM starts to do a new exploratory search.

We start to consider the case in which the first variable



**Figure 2. Example in which  $c_0 - b_0 = 15$ . In that case we will have  $b_s = c_0$ .**

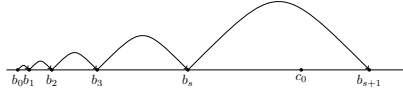


**Figure 3. Example in which  $c_0 - b_0 = 17$ . In that case  $b_{s+1}$  is not accepted, and that is represented by a dashed arrow.**

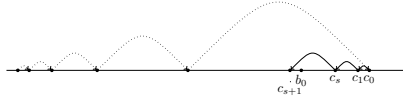
to be searched for is  $b$ . It cannot be decreased (otherwise the fitness would be worse), and it has a gradient to increase toward  $c$ . In the best case, there will be  $\Omega(\log(c - b))$  steps, because in the pattern search the step size doubles each time. Considering that  $E[c - b|a] = E[c - a|b]/2 = \Theta(n)$ , then the number of expected steps is  $\Omega(\log n)$ . Let  $s$  be the number of steps for which we get as close to  $c$  as possible. After  $s$  steps, the increment to the variable  $b$  is  $\sum_{i=0}^s 2^i = 2^{s+1} - 1$ . Let  $k = 2^{s+1}$ , hence after  $s$  steps the value of  $b$  is  $b_s = b_0 + k - 1$ . Figure 2 shows an example in which  $b_s = c_0$ . Unfortunately,  $b_s$  can be different from  $c_0$ , but we can be sure that  $b_{s+1} > c_0$ . Two opposite cases requires to be studied: whether for  $b_{s+1} = b_0 + 2k - 1$  we get a better fitness or not. If we get a worse fitness (as shown in an example in figure 3), we will have a new exploratory search around  $b_s$  (that now has become the new  $b_0$ ), and it will have a gradient to increase toward  $c$ . We can inductively apply the same discussion on the new value  $b_0$ , with the difference that the distance from  $c$  has been decreased. In particular, the original distance  $d = c_0 - b_0$  (with  $b_0$  the old value, and not the new one that is equal to  $b_s$ ) is decreased down to  $d = c_0 - b_s$ . To note that the distance of  $b$  from  $c$  influences the fitness in a linear way. In the best case we would have  $d = 1$ , but what happens in the worst case? In other words, what is the lowest value that  $b_s$  can have? The following inequalities must be satisfied:

$$\begin{cases} b_0 + k - 1 < c_0, \\ b_0 + 2k - 1 > c_0, \\ c_0 - b_s \leq b_{s+1} - c_0, \end{cases}$$

which is the case for  $k \geq \frac{2}{3}(c_0 - b_0 + 1)$ . Hence, even for the lowest  $b_s$ , the distance from  $c$  is reduced by at least  $\frac{2}{3}$ . The opposite case is when  $b_{s+1}$  is accepted (an example is shown in figure 4). In that case it would become the new  $c_0$ .



**Figure 4. Example in which  $c_0 - b_0 = 25$ . In that case  $b_{s+1}$  is accepted.**



**Figure 5. Example in which  $c_0 - b_0 = 6$ . In that case  $c_{s+1}$  is accepted. To note that is the continuation of the search done in figure 4, which is represented in dotted arrows. The former  $b_{s+1}$  has become the new  $c_0$ , whereas the old  $c_0$  is now the new  $b_0$ .**

We are interested in the highest value that  $b_{s+1}$  can take. Given the inequalities:

$$\begin{cases} b_0 + k - 1 < c_0, \\ b_0 + 2k - 1 > c_0, \\ c_0 - b_s > b_{s+1} - c_0, \end{cases}$$

the highest value that we can have is  $b_{s+1} = b_0 + \frac{4}{3}(c_0 - b_0 + 1)$ . Therefore, the distance of  $b_0$  from  $c_0$  has been reduced again at least by  $\frac{2}{3}$ . That means that, whatever the situation is, we can reduce the distance  $c_0 - b_0$  by  $\frac{2}{3}$  in  $\Theta(\log n)$  steps. Before analysing the new exploratory search starting from either  $b_s$  or  $b_{s+1}$  (that will become the new  $c_0$ ), we need to analyse the case in which the considered variable for the exploratory search is  $c$ . The case in which the considered variable is  $c$  has the same specular properties as  $b$ . The variable cannot be increased (Proposition 2), and it has a gradient to decrease down to  $b_0$ . Figure 5 shows an example in which  $c$  decreases. That example is the continuation of the search in figure 4.

If  $c$  does not arrive down to  $b_0$  in a single pattern search, that will stop with a distance from  $b_0$  that has been reduced by at least  $\frac{2}{3}$ . The only difference is that  $c$  can decrease so much that  $c_{s+1}$  might become lower than  $a_0$ . In that case,  $c_{s+1}$  would become the new  $a_0$ , with a gradient toward  $b_0$  (that has become the new  $c_0$ ). Even if  $c_{s+1} < a_0$ , its distance from  $b_0$  would be no more than  $\frac{1}{3}$  of the original distance  $c_0 - b_0 + 1$ .

The case in which the considered variable is  $a$  follows the same behaviour of  $b$ , with the difference that before reaching  $c$  it will become the new  $b$  as soon as it gets bigger

than the starting  $b_0$ .

If we start from either  $b$  or  $c$ , a pattern search will reduce the distance  $c - b$  by at least  $\frac{2}{3}$  in  $\Theta(\log n)$  steps. How many new exploratory searches followed by pattern searches do we need before getting  $b = c$ ? In the worst case we get the distance  $c - b$  reduced by only  $\frac{2}{3}$ , hence we need  $O(\log n)$  searches. That would mean that we get  $b = c$  in  $\Omega(\log n)$  and  $O((\log n)^2)$  steps. However, the distance  $c - b$  gets reduced at each search. Hence, the upper bound for the steps is:

$$\begin{aligned} \sum_{i=0}^{\log n} \log\left(\frac{n}{3^i}\right) &= \sum_{i=0}^{\log n} (\log n + \log 3^{-i}) \\ &= (\log n)^2 - \log 3 \sum_{i=0}^{\log n} i \\ &= (\log n)^2 - \log 3 \frac{(\log n)(\log n + 1)}{2} \\ &= O((\log n)^2). \end{aligned}$$

Unfortunately, although the distance is reduced at each new pattern search, the upper bound does not change.

Once we get  $b = c$ , the way  $a$  changes follows the same discussion done for  $b$  and  $c$ . Note that, if we start from  $a$ , after a certain point it will become  $b$ . Therefore, from whatever variable (i.e.,  $x$ ,  $y$  or  $z$ ) AVM starts the search, we expect to get  $b = c$  in  $\Omega(\log n)$  and  $O((\log n)^2)$  steps, and then the final  $a = b = c$  in additional  $\Omega(\log n)$  and  $O((\log n)^2)$  steps, which does not change the asymptotic runtime.

To finish the proof, we still need to consider the case in which  $a + b > c$ . It follows the same type of proof of  $a + b \leq c$ . The only difference is that  $b$  can be changed only in two cases:

$$\begin{aligned} b &= a + 1, \\ b &= c - 1. \end{aligned}$$

In these two cases, a single exploratory search makes either  $a = b$  or  $b = c$  in at most two steps. The final  $a = b = c$  will be reached in  $\Omega(\log n)$  and  $O((\log n)^2)$ . As stated above, a constant number of restarts suffices to find a global optimum.  $\square$

**Theorem 7.** *The probability that AVM has found an optimal solution to TC within  $c \cdot n \cdot (\log n)^2$  iterations is exponentially large  $1 - e^{-\Omega(n)}$ , where  $c$  is a constant.*

*Proof.* Except for the choice of search point in the initial iteration, or in case of a restart, AVM is a deterministic algorithm. By the proof of Theorem 6, AVM has reached a local optimum within  $c \cdot (\log n)^2$  iterations, for some constant  $c$ . A global optimum is found if the initial search point satisfies  $a > 0$ , an event which occurs with constant probability  $p$ . The probability that AVM needs more than  $n$  restarts before the initial search point satisfies the condition above, is less than  $(1 - p)^n = e^{-\Omega(n)}$ .  $\square$

## 5 Conclusions and Future Work

In this paper we have theoretically analysed the runtime of three search algorithms (RS, HC and AVM) on the test data generation for the TC problem. Our theoretical analyses confirm the experimental results obtained so far in the literature. In other words, on that problem RS has a worse time complexity than HC, and AVM has the best.

At any rate, the most used meta-heuristic in software testing are GAs. Hence, we are currently investigating its runtime. The problem is that, in literature, GAs have been very difficult to analyse.

Considering the smooth nature of the search landscape, we would expect that GAs would have a complexity not better than the one of HC, and not worse than the one of RS. The question would be whether the runtime complexity of GAs would be closer to RS or to HC.

However, experimental results and wrong assumptions can be misleading. Therefore, it might be possible that GAs have an inferior runtime to RS, but with a lower constant which hides this fact in experimental results. In the same way, although GAs are global search algorithms, bigger jumps done by the crossover operator might make the runtime of GAs better than that of HC. Therefore, we need to theoretically prove the runtime complexity of GAs to shed light on these questions.

For the future we are also planning to analyse more complex software. Moreover, it will be important to analyse how the obtained results can be generalised for different classes of test problems.

## 6 Acknowledgements

The authors are grateful to Ramón Sagarna for insightful discussions. This work is supported by EPSRC grants EP/D052785/1 and EP/C520696/1.

## References

- [1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.
- [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] R. A. DeMillo, R. J. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
- [4] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276:51–81, 2002.
- [5] 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.
- [6] F. Gruenberger. Program testing: The historical perspective. *Program Test Methods*, pages 11–14, 1973.
- [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] J. H. Holland. *Adaptation in Natural and Artificial Systems, second edition*. MIT Press, Cambridge, 1992.
- [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 formal models: Research articles. *Software Testing, Verification and Reliability*, 14(2):81–103, 2004.
- [12] P. K. Lehre and X. Yao. Runtime analysis of (1+1) ea on computing unique input output sequences. In *Congress on Evolutionary Computation (CEC)*, 2007.
- [13] J. C. Lin and P. L. Yeh. Automatic test data generation for path testing using GAs. *Information Sciences*, 131(1-4):47–64, 2001.
- [14] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.
- [15] 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.
- [16] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.
- [17] 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(1):100–106, 2007.
- [18] C. V. Ramamoorthy, S. B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.
- [19] G. Tasse. The economic impacts of inadequate infrastructure for software testing, final report. *National Institute of Standards and Technology*, 2002.
- [20] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.
- [21] M. Woodward. Editorial: A test of time across the generations. *Software Testing, Verification & Reliability*, 14(2):79–80, 2004.
- [22] 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.