2010 Third International Conference on Software Testing, Verification and Validation

It Does Matter How You Normalise the Branch Distance in Search Based Software Testing

Andrea Arcuri
Simula Research Laboratory, P.O. Box 134, Lysaker, Norway.
arcuri@simula.no

***Abstract*—The use of search algorithms for test data generation has seen many successful results. For structural criteria such as branch coverage, heuristics have been designed to help the search. The most common heuristic is the use of *approach level* (usually represented with an integer) to reward test cases whose executions get close (in the control flow graph) to the target branch. To solve the constraints of the predicates in the control flow graph, the *branch distance* is commonly employed. These two measures are linearly combined. Because the approach level is more important, the branch distance is *normalised*, often in the range [0,1]. In this paper, we analyse different types of normalising functions. We found out that the one that is usually employed in the literature has several flaws. We hence propose a different normailising function that is very simple and that does not suffer of these limitations. We carried out empirical and analytical analyses to compare these two functions. In particular, we studied their effect on two commonly used search algorithms, namely Simulated Annealing and Genetic Algorithms.**

***Keywords*-Branch Distance, Search Based Software Testing, Theory, Simulated Annealing, Genetic Algorithms, Test Data Generation.**

I. Introduction

Software testing is an expensive activity. There are different types of testing, and white box testing is a common practice [20]. The goal is to obtain test suites that once executed maximise the coverage of the software under test (SUT). A set of test cases that exercises most parts of the SUT is usually considered better then a set that only covers few parts. In fact, faults can lie in specific parts of the SUT, and we need to execute those parts with some test cases to reveal those faults.

There are different types of coverage criteria, such as statement coverage, branch coverage and path coverage. In the rest of the paper we will deal only with branch coverage, because is likely the most used. Our analyses could be extended to other coverage criteria as well. In branch coverage, we want to cover all possible branches in the control flow of the SUT. For example, for each "if" statement, we want to obtain test cases that execute both its "then" and its "else" branch. Not all the branches are feasible, so obtaining 100% coverage could be not possible in some cases.

Often, obtaining a set of test cases that maximises the coverage of the SUT is not trivial. Complex computations, difficult control and data flows and non-linear predicates can make hard to produce such test cases. Considering all the possible test cases is impractical, because most of the time the domain of test cases is infinite. Therefore, lot of effort has been spent in designing novel techniques to automate this software engineering task. Random testing [4] is a cheap and easy technique that can obtain reasonable coverage, but it fails to cover difficult branches. Another common technique is for example *symbolic execution* [11]. Successful applications have been reported in the literature, but still symbolic execution has scalability problems and for example it still has issues in handling complex data structures and pointers.

Test data generation for software coverage can be considered as an optimisation problem, so *search algorithms* can be used to tackle it [7]. Search algorithms have been applied to tackle many software engineering problems [8], and particularly software testing [15].

One application of search algorithms in software testing is the search for test cases that cover particularly difficult branches. To guide the search for these test cases, heuristics are used to define a *fitness function*. A fitness function is used to compare different test cases, and it gives a score of how "good" a test case is. A test case that gets "closer" to execute the target branch will have a better fitness value. Search algorithms use the fitness function to focus their search on promising areas of the search space.

There are two main heuristics used for the fitness function, i.e. the *approach level* and the *branch distance* (we will discuss them in more details later in the paper) [15]. For the fitness function, these two measures are linearly combined together. The approach level is usually an integer value. Because the branch distance is less important than the approach level, it is usually *normalised* in the range $[0,1]$. This guarantees that a better approach level is always preferred regardless of the branch distance measure.

A normalising function $\omega : \Re^+ \rightarrow [0,1]$ should guarantee that $x_i < x_j \Leftrightarrow \omega(x_i) < \omega(x_j)$ and $x_i = x_j \Leftrightarrow \omega(x_i) = \omega(x_j)$, i.e. the order of the scaled values has to be the same. Different normalising functions exist, but they can have different properties.

In this paper, we analyse the role of the normalising function in the context of search based software testing. We

978-0-7695-3990-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICST.2010.17

criticise the normalising function that is commonly used in the literature. It is computationally expensive, and it is very prone to arithmetic and precision errors. We hence propose to use a different normalising function that does not suffer of these problems.

We then analytically compare these two functions on commonly used search algorithms such as Simulated Annealing and Genetic Algorithms. We formally prove that, in the case of Simulated Annealing, the choice of the normalising function has strong effect on its *runtime* (i.e., the number of fitness evaluations it requires before finding an optimal solution). In the case of Genetic Algorithms, there is not much difference as long as the parameters of the normalising functions are properly chosen.

For sake of clarity, we carried out a set of experiments on a toy problem and simulations to explain the consequences of our analytical results. However, it is important to note that our analytical results are general and independent from the analysed case study. In the same way, the computational cost and the precision errors of the normalising functions are independent as well from the case study. In other words, the toy problem we use for the empirical analysis has only a didactic motivation. Its simplicity allows us to give precise reasons for the behaviour of search algorithms applied to it. Nevertheless, this does not undermine the generality of our results.

This paper gives the following contributions to search based software testing:

- The role of the normalising function has been practically ignored in the literature. To the best of our knowledge, this is the first work that analyses it in details.
- We criticise the normalising function that is commonly used in the literature. It has several flaws, as for example side-effects on the runtime of Simulated Annealing. We hence propose a different function that is very simple and does not suffer of those problems.
- Normalising functions are used practically in all the testing tools that employ search algorithms. Changing the normalisng function would consist of only changing few lines in those tools. Therefore, our results are very easy to apply in practice and have wide scope in most of the search based software testing applications.
- Search based software engineering lacks of theoretical foundations [1]. The analyses in this paper are a further contribution in this area.

The paper is organised as follows. Section II describes in details the fitness function that is commonly used for branch coverage. In Section III we discuss the normalising functions used in this paper. Brief background information on the analysed search algorithms is provided in Section IV. Our analyses are presented in Section V. Section VI discusses the threats to validity of our work. Finally, Section VII concludes the paper.

II. Fitness Function

To apply search algorithms to test data generation for branch coverage, we need to define a fitness function f to evaluate the test cases. In our case, we need to minimise f. If the target branch is covered when a test case t is executed, then $f(t) = 0$, and the search is finished. Let's assume that a test case t is composed of only a set of inputs I, i.e. $f(t) = f(I)$ (in general a test case could be more complex, because it could contain a sequence of function calls to put the internal state of the SUT in the appropriate configuration). Otherwise, it should heuristically assign a value that tells us how far the branch is from being covered. That information would be exploited by the search algorithm to reward "better" solutions.

The most famous fitness function is based on two measures: the *approach level* $\mathcal{A}$ and the *branch distance* δ. The measure $\mathcal{A}$ is used to see in the data flow graph how close a computation is from executing the node on which the target branch depends on. The branch distance δ heuristically evaluates how *far* a predicate is from obtaining its opposite value. For example, if the predicate is true, then δ tells us how far the input I is from an input that would make that predicate false.

For a target branch z, we have that the fitness function f_z is:

$$f_z(I) = \mathcal{A}_z(I) + \omega(\delta_{pred(y)}(I)) \ . \tag{1}$$

Notice that the branch distance δ is calculated on the node of *diversion*, i.e. the last node in which a critical decision (not taking the branch y) is made that makes the execution of z not possible (i.e., $pred(y)$ represents the predicate in the node that we need to satisfy to execute the branch y). For example, branch z could be nested to a node N (in the control flow graph) in which branch y 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 $\delta_{pred(y)}$ to enter in the nested branches.

Let $\{N_0, \ldots, N_k\}$ be the sequence of diversion nodes for the target z, with N_i nested to $N_{j>i}$. Let D_i be 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 $\mathcal{A}_z(I_{i+1}) = \mathcal{A}_z(I_i) + c$, where c can be any positive constant (e.g., $c = 1$) 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 δ with a normalising function ω such that $0 \leq \omega(\delta_{pred(y)}) < c$ for any predicate. Notice that δ is never negative. We need to guarantee that the order of

Table I
EXAMPLE OF HOW TO APPLY THE FUNCTION δ ON SOME PREDICATES. k CAN BE ANY ARBITRARY POSITIVE CONSTANT VALUE. A AND B CAN BE ANY ARBITRARY EXPRESSION, WHEREAS a AND b ARE THE ACTUAL VALUES OF THESE EXPRESSIONS BASED ON THE VALUES IN THE INPUT SET I.

Predicate θ	Function $\delta_\theta(I)$
A	if a is TRUE then 0 else k
$A = B$	if $abs(a-b) = 0$ then 0 else $abs(a-b)+k$
$A \neq B$	if $abs(a-b) \neq 0$ 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 $(a-b)+k$
$A > B$	$\delta_{B<A}(I)$
$A \geq B$	$\delta_{B \leq A}(I)$
$\neg A$	Negation is moved inward and propagated over A
$A \wedge B$	$\delta_A(I) + \delta_B(I)$
$A \vee B$	$min(\delta_A(I), \delta_B(I))$

the values does not change once mapped with ω, for example $x_0 > x_1$ should imply $\omega(x_0) > \omega(x_1)$.

How the branch distance is defined? Because it is an heuristic, there can be different definitions. Table I shows a possible definition. The branch distance δ_θ takes as input a set of values I, and it evaluates the expressions in the predicate θ based on the actual values in I. This function δ works fine for expressions involving numbers (e.g., integer, float and double) and boolean values. For other types of expressions, such as an equality comparison of pointers/objects, we need to define the semantics of the subtraction operator $-$. For example, it can return 1 if the two values are different and 0 otherwise. More details can be found in [15].

To ease the notation, in the rest of the paper we will omit to specify the branches y for δ and z for $\mathcal{A}$. They will implicitly refer to the target branch we want to cover.

III. NORMALISING FUNCTION

Several normalising functions exist. In this paper we consider the one that is usually employed in the literature (ω_0) and another simple one (ω_1) that we suggest to use instead of ω_0. They are:

$$\omega_0(x) = 1 - \alpha^{-x} \ ,$$

$$\omega_1(x) = \frac{x}{x+\beta} \ ,$$

where $\alpha > 1$ and $\beta > 0$ are constants that need to be set. For both these two functions, it is guaranteed that the order of inputs is preserved in the mapping. Given $\epsilon > 0$, in fact we have:

$$\omega_0(x+\epsilon) = 1-\alpha^{-(x+\epsilon)} = 1-(\alpha^{-x} \cdot \frac{1}{\alpha^\epsilon}) > 1-\alpha^{-x} = \omega_0(x) \ ,$$

```
public static double omega0(double x){
       return 1.0d - Math.pow(alpha, -x);
}

public static double omega1(double x){
       return x / (x + beta);
}
```

Figure 1. Java code for the normalising functions.

$$\begin{aligned} \omega_1(x+\epsilon) &= \tfrac{x+\epsilon}{x+\epsilon+\beta} \\ &= \tfrac{x+\epsilon}{x+\epsilon+\beta} + \tfrac{\beta}{x+\epsilon+\beta} - \tfrac{\beta}{x+\epsilon+\beta} \\ &= 1 - \tfrac{\beta}{x+\epsilon+\beta} \\ &> 1 - \tfrac{\beta}{x+\beta} \\ &= \tfrac{x}{x+\beta} = \omega_1(x) \ . \end{aligned}$$

Furthermore, $\omega_0(0) = \omega_1(0) = 0$. The function ω_0 was first proposed by Baresel [3], and it has been then used in several successive studies (e.g., [16], [9]). A typical value is $\alpha = 1.001$. However, often the normalising function is considered a minor component of the system. Although it is used, the employed normalising function is often not even specified (e.g., [23], [17]).

These two functions are easy to code. Figure 1 shows possible implementations in Java. Notice that those implementations are simple, and they do not take into account possible optimisations. For example, if they are called always on a restricted set of inputs, a pre-computed look-up table would be more efficient.

IV. SEARCH ALGORITHMS

There exist several search algorithms. There is no search algorithm that is best on all search problems [25]. Once a specific class of problems is considered (e.g., software engineering), it is possible that a particular search algorithm performs better than the others on that class. On a new search problem (e.g., software testing) is hence important to compare different search algorithms [2].

In the literature, Simulated Annealing and Genetic Algorithms are the search algorithms most used to tackle software engineering problems [8]. They have been reported to produce successful results in many applications. Therefore, we focus our analyses to these two search algorithms. In this section we give brief background information about them.

A. Simulated Annealing

Simulated Annealing [12] is a search algorithm that is inspired by a physical property of some materials used in metallurgy. Heating and then cooling the temperature in a controlled way often brings to a better atomic structure. In fact, at high temperature the atoms can move freely, and a slow cooling rate makes them to be fixed in suitable positions. In a similar way, a temperature is used in Simulated

Annealing to control the probability of moving to a worse solution in the search space. The temperature is properly decreased during the search.

Simulated Annealing is similar to local search. It keeps one solution ρ at the time and then at each step it samples a new neighbour ρ_n. A neighbour is a solution that is structurally close. For example, in a binary representation, all solutions that differ by only one bit can be considered neighbours. If this neighbour has better fitness value (i.e., $f(\rho_n) < f(\rho)$), then Simulated Annealing moves to it. Otherwise, it moves to that worse solution only accordingly to a probability P_{ρ_n} that is based on the current temperature T:

$$P_{\rho_n} = e^{-\frac{f(\rho_n)-f(\rho)}{T}} \ . \tag{2}$$

If all the neighbour solutions have equal or worse fitness value, then ρ is a *local optimum*. Simulated Annealing needs to be able to move to worse solutions in order to escape from the local optima.

Different strategies can be defined to decrease the temperature T (i.e., the cooling schedule). A simple one would be to modify it with $T_{i+1} = \eta T_i$ (i.e., geometric cooling) where $\eta < 1$ and i is the time when the updating takes place. This cooling can be applied every time after a constant number of iterations (e.g., every 1000 fitness evaluations).

Simulate Annealing has been used for example for test data generation for coverage criteria [22], [14], regression testing [13] and for testing finite state machines [6].

B. Genetic Algorithms

Genetic Algorithms [10] are the most famous metaheuristic used in the literature of search based software engineering [8]. They are inspired by the Darwinian Evolution theory. They rely on four basic features: *population*, *selection*, *crossover* and *mutation*. More than one solution is considered at the same time (*population*). At each generation (i.e., at each step of the algorithm), some good solutions in the current population chosen by the selection mechanism generate offspring using the crossover operator. This operator combines parts of the chromosomes (i.e., the solution representation) of the offspring with a certain probability, otherwise it just produces copies of the parents. These new offspring solutions will fill the population of the next generation. The mutation operator is applied to make small changes in the chromosomes of the offspring. To avoid the possible loss of good solutions, a number of best solutions can be copied directly to the new generation (*elitism*) without any modification.

Fitter individuals should have more chances to survive and reproduce. This is represented by the selection mechanism. There are different types of selection mechanism. In this paper we only consider "fitness proportional selection" (also called "roulette wheel selection"). It is practically the first proposed selection mechanism [10]. It is used in software testing (e.g., [26]) and it is still subject of formal analyses (e.g., [21]).

Given a population G with size $|G| = m$, each individual $g \in G$ has fitness value $f(g)$. In the case of minimising fitness functions (as we have in this work), the probability p_k of choosing an individual g_k for reproduction is:

$$p_k = \frac{\frac{1}{v+f(g_k)}}{\sum_{g \in G} \frac{1}{v+f(g)}} \ , \tag{3}$$

where v is a constant that in general has value 1 (see for example [5]). For lower (i.e., better) fitness values, g_k obtains higher probability p_k of being selected. Notice that, if all solutions have same fitness values, then $p_k = 1/m$. This would be the same probability if we were not rewarding fitter individuals, i.e. all individuals would have the same chances of reproduction.

Notice that in fitness proportional selection the actual row values of the fitness function are used, so how the branch distance is normalised has effect on Equation 3. On the other hand, in other popular selection mechanisms in which the fitness values are used only for direct comparisons (e.g., tournament and rank selection) the normalising distance plays no role, apart from its computational cost and rounding errors (both discussed in more detailed in Section V-A).

V. Analysis of Normalising Functions

In this section we analyse which differences there can be if we use the normalising function ω_1 instead of the traditional ω_0. We first discuss their computational cost and their robustness to arithmetic errors and loss of precision. Then, we discuss their effect on the runtime of Simulated Annealing and Genetic Algorithms.

A. Cost and Precision

Often, the most time consuming component of a search system is the evaluation of the fitness function. In software testing, the execution of a test case could be particularly time consuming. So, inefficiency in the normalising function could be just ignored. However, for the computation of ω_0, a power operation needs to be computed. Based on the implementation given in Figure 1, we need to analyse the source code of "Math.pow". It calls a native method, whose C implementation is taken from the "Freely Distributable Math Library" *fdlibm* (see Java documentation for more details). It is a complex method that is longer than 200 lines of code. Its computational cost is hence much higher than the cost of computing ω_1.

For a normalising function ω, it is important that $x_i < x_j \Leftrightarrow \omega(x_i) < \omega(x_j)$. Both ω_0 and ω_1 satisfy this property. But their implementation in a programming language is necessarily prone to precision errors. Not all numbers can be represented in a computer with finite memory. In many

languages, "double" values consist of 64 bits. Although they can represent 2^{64} possible values, problems can still arise.

To validate the robustness of these normalising functions, we have designed a simple technique to compare them. We considered integer inputs x_i starting from 0, and we checked the smallest value for which we have an error, i.e. $x_i < x_{i+1} \wedge \omega(x_i) \geq \omega(x_{i+1})$. Notice that we used integers instead of continuos values (e.g., "double" and "float" types) because it was easier to enumerate them. This experiment was carried out on the implementation of normalising functions depicted in Figure 1. For these normalising functions, we used the values $\alpha = 1.001$ (which is the suggested value in the literature) and $\beta = 1$ (which we used in a previous work [2]). On a Java Virtual Machine version 6, for ω_0 we get the first error for $x_i = 29{,}877$. On the other hand, ω_1 is far more robust. In fact we get the first error only for $x_i = 94{,}911{,}151$.

B. Simulated Annealing

In this section we study whether the choice of ω_1 instead of ω_0 can have effect on Equation 2. In other words, we study whether the probability of moving to a worse neighbour can be significantly effected by the normalising function.

Let ρ be the current solution that Simulated Annealing is considering, with $f(\rho) > 0$ (i.e., ρ is not an optimal solution). Let ρ_n be the neighbour solution that is sampled, with $\mathcal{A}(\rho_n) = \mathcal{A}(\rho)$ and $\sigma(\rho_n) = \sigma(\rho) + \epsilon$, where $\epsilon > 0$. In other words, ρ_n has the same approach level but it has a worse branch distance. This can be considered as a common scenario.

To simplify the notation, for Equation 2 we use the variable $\gamma = e^{-1/T} < 1$. If we use the normalising function ω_0, we obtain that the probability $P^{\omega_0}_{\rho_n}$ to accept the new worse solution ρ_n is:

$$\begin{aligned} P^{\omega_0}_{\rho_n} &= \gamma^{(\mathcal{A}(\rho_n)+\omega_0(\delta(\rho_n)))-(\mathcal{A}(\rho)+\omega_0(\delta(\rho)))} \\ &= \gamma^{\omega_0(\delta(\rho)+\epsilon)-\omega_0(\delta(\rho))} \\ &= \gamma^{1-\alpha^{-(\delta(\rho)+\epsilon)}-(1-\alpha^{-\delta(\rho)})} \\ &= \gamma^{\alpha^{-\delta(\rho)}(1-\alpha^{-\epsilon})} \ . \end{aligned} \tag{4}$$

On the other hand, for ω_1 we obtain:

$$\begin{aligned} P^{\omega_1}_{\rho_n} &= \gamma^{(\mathcal{A}(\rho_n)+\omega_1(\delta(\rho_n)))-(\mathcal{A}(\rho)+\omega_1(\delta(\rho)))} \\ &= \gamma^{\omega_1(\delta(\rho)+\epsilon)-\omega_1(\delta(\rho))} \\ &= \gamma^{\frac{\delta(\rho)+\epsilon}{\delta(\rho)+\epsilon+\beta}-\frac{\delta(\rho)}{\delta(\rho)+\beta}} \\ &= \gamma^{\frac{\beta\epsilon}{(\delta(\rho)+\beta)(\delta(\rho)+\epsilon+\beta)}} \ . \end{aligned} \tag{5}$$

These two probabilities $P^{\omega_0}_{\rho_n}$ and $P^{\omega_1}_{\rho_n}$ are very different. In both, the actual row branch distance of the current solution ρ has effect on them. But in $P^{\omega_0}_{\rho_n}$ it has an exponential effect (i.e, $\alpha^{-\delta(\rho)}$), whereas in $P^{\omega_1}_{\rho_n}$ it has a polynomial effect (i.e. $\approx \delta(\rho)^{-2}$). This can completely change the *runtime* of Simulated Annealing. Based on the choice ω_0 or ω_1, the number of fitness evaluations done before reaching an optimal solution will be different. In fact, with ω_0 we obtain higher probabilities of accepting worse solutions for higher $\delta(\rho)$ compared to ω_1.

Figure 2 plots $P^{\omega_0}_{\rho_n}$ ($\alpha = 1.001$) whereas Figure 3 plots $P^{\omega_1}_{\rho_n}$ ($\beta = 1$). The difference in the branch distance is $\epsilon = 1$. These probabilities are plotted based on different values of the temperature T and the branch distance $\delta(\rho)$ of the current solution. Both these variables are considered with the values 10^i where $i \in \{-5, -4, \ldots, 3, 4\}$. As we can see, the two graphs are significantly different. Most of the time, the value of $P^{\omega_0}_{\rho_n}$ is very close to 1.

We analysed how this difference can effect the performance of Simulated Annealing on a toy problem. Figure 4 shows a simple program with a single branch. The target is executing the "then" branch. The predicate $x == 0$ directly depends on the input x. Let's assume we are going to test only positive values. In this case, $\mathcal{A} = 0$ and $f(x) = \delta(x) = x$.

Let's consider a Simulated Annealing in which a neighbour is sampled by randomly adding ± 1 to the current solution. In other words, with probability $1/2$ we add 1, otherwise we subtract 1. The starting solution is randomly chosen in an interval $\{0, \ldots, k\}$, where k can be for example 100 or 1000 (notice that constraints on the input variables are common in the literature [15]).

Under these assumptions, the testing of this toy program is equivalent to the problem called *Gambler's Ruin* [19], which is a very well known and studied problem with practical applications in economics. The variable x would represent the amount of money the gambler has. The neighbourhood ± 1 would be the bet (e.g., 1 euro). Losing the bet would have probability $1/2$. Winning a bet would have probability $Win = (1/2)P_{\rho_n}$. With probability $(1/2)(1 - P_{\rho_n})$ the gambler keeps his bet. The target $x == 0$ would be achieved when the gambler loses all of his money. However, our case is slightly more complicated then the basic gambler's ruin problem. In fact, Win is not a constant because it depends on the current amount of money (i.e., the input x, see $\delta(\rho)$ in Equation 4 and 5) and on the time elapsed (in our context, it would be represented by the cooling temperature).

In this problem, there is not any local optimum. Therefore, moving to worse neighbours is always a bad choice. Having a temperature $T \to 0$ would be the optimal setting. However, we are not interested to find the best setting to solve this toy problem with Simulated Annealing. We are in fact interested in studying the difference in the runtime based on the choice of the normalising function with the same setting of the temperature and the cooling schedule. Notice that for some problems it is better to have a higher probability of accepting worse solutions (e.g., there could be many local optima), whereas in other problems it could be the opposite. Hence, it is not important whether on this toy problem ω_1 is better

than ω_0 or vice-versa.

We ran a set of experiments in which the starting temperature is set to 1000, and every 1000 iterations it is reduced by 10% (i.e., $\eta = 0.9$). These are arbitrary settings that resemble realistic choices. Notice that for each search problem, the optimal settings are problem dependent and time needs to be spent to tune them.

We chose different starting values x ranging from 1 to 1024 (notice that for too high values we would end up in the precision errors described in Section V-A). We compared Simulated Annealing when ω_0 and ω_1 are used. For comparison, we also considered the case when worse solutions are always accepted. For each value of x, we ran 100 experiments with different random seeds. Figure 5 shows the average number of fitness evaluations before reaching the optimal solution $x = 0$. For this testing problem, there is a clear difference between ω_0 and ω_1. By simply changing the normalising function, we can obtain for example a double speed up.

Simulated Annealing bases the probability of moving to a worse solution ρ_n on its difference d in energy (i.e., the fitness value) with the current solution ρ, i.e. $d = f(\rho_n) - f(\rho)$. It follows $f(\rho_n) = f(\rho) + d$. If their difference is for example $d = 1$, it does not matter whether $f(\rho)$ is equal to 1 or to 1000 ($f(\rho_n)$ would be equal to 2 and 1001 respectively). The probability of accepting a worse solution P_{ρ_n} would not change, because d does not change.

When the approach level is 0, the fitness depends only on the branch distance. To use Simulated Annealing in the way it was developed for, we would use the branch distance values directly, with $d = \delta(\rho_n) - \delta(\rho) = \epsilon$. In other words, a difference ϵ in the branch distance should be reflected in a difference d in energy that should be considered in the same way regardlessly of the current value of the energy. However, the use of the considered normalising functions breaks this property. As we have shown in Equation 4 and 5, the current energy effects the probability P_{ρ_n} of moving to a worse solution. This is an unwanted side effect. Whether there should be higher or lower probability P_{ρ_n} is something that should be decided in the temperature T and in its cooling schedule. It should not be a side effect of the choice of the normalising function. Because for ω_1 this effect of the current energy has a polynomial nature whereas for ω_0 it is exponential, we argue that this is a further reason to prefer ω_1.

Notice that there will be different optimal cooling schedules for ω_0 and ω_1. Finding these schedules is problem dependent, and it requires experimentation to tune them. Studying whereas the choice of ω_0 and ω_1 makes this process easier or more difficult cannot be done in general without considering specific case studies. It is an interesting research question that we will address in future work.

It could be tempting to use the following normalising function ω_2:

```
public void foo(int x) {
  if(x==0){
    // TARGET
  }
}
```

Figure 4. Toy program.

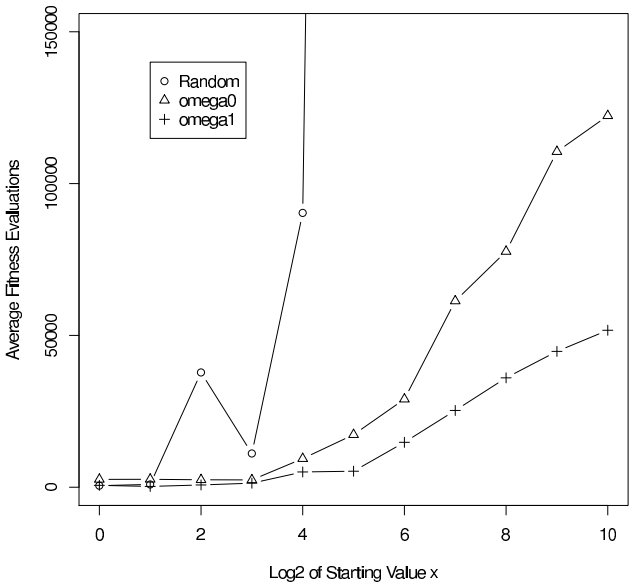

Figure 5. Average number of fitness evaluations to find input $x = 0$.

$$\omega_2(x) = \frac{x}{q} \ ,$$

where q is a constant. In this case:

$$\omega_2(\delta(\rho_n)) - \omega_2(\delta(\rho)) = \frac{\delta(\rho) + \epsilon}{q} - \frac{\delta(\rho)}{q} = \frac{\epsilon}{q} \ .$$

For ω_2, it *seems* that the current value $\delta(\rho)$ disappears. But to guarantee $\omega_2(x) < 1$ for all possible inputs, it should always be $q > \delta(\rho)$. In other words, it does not exist a constant q for which ω_2 would be a valid normalising function. Choosing the highest possible value for q would have three main issues. First, ω_2 would still be invalid and theoretically unsound. Second, a division with the highest possible value the used programming language handles would likely end up with serious rounding errors (we have not tested this hypothesis though). Finally, the values of the energy would be extremely low. Unless extremely low temperatures would be used, we would end up with $P_{\rho_n} \approx 1$ during all the search process.

C. Genetic Algorithms

Genetic Algorithms are complex search techniques in which many factors play a role on the final outcome (e.g.,

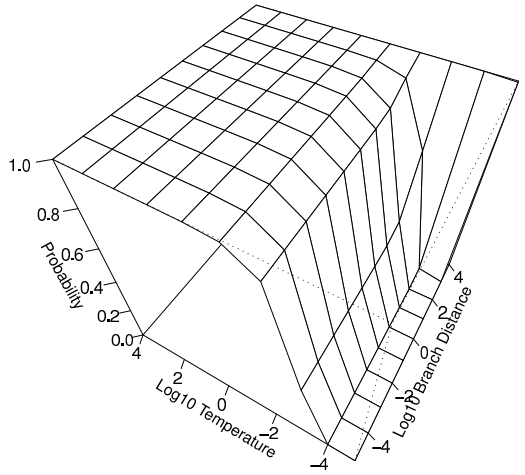

Figure 2. Probability of accepting a worse solution when ω_0 is used.

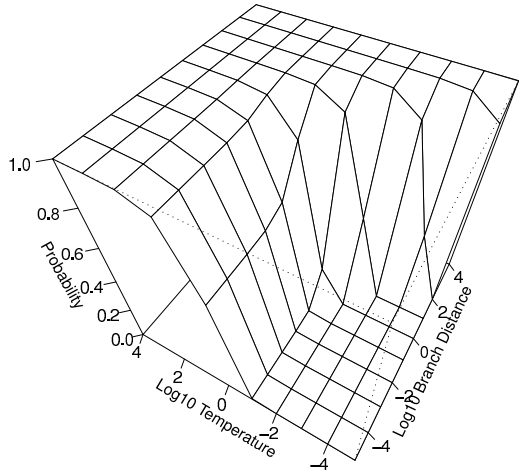

Figure 3. Probability of accepting a worse solution when ω_1 is used.

population size and mutation rate). In this paper, we focus on the fitness proportional selection mechanism (see Equation 3). In fact, as we will show, the setting of the normalising function ω can have large impact on the performance.

Let's consider a Genetic Algorithm with a population G of individuals with size $|G| = m$. If the selection mechanism does not reward the solutions based on their fitness values, then an unbiased selection mechanism should choose individuals with same probability $1/m$. Let p_b the probability of selecting the best individual b in the population, whereas p_w is the probability of choosing the worst individual w, i.e. $w \in G$ and $f(w) \geq f(g) \forall g \in G$. In a *reasonable* selection mechanism, we would expect $p_b \geq 1/m$ and $p_w \leq 1/m$. Their actual values depend on the *selection pressure* of the selection mechanism.

Notice that all the probabilities are always bounded from above by the value 1, but for simplicity we do not specify it in each formula.

Independently of the normalising function ω and the SUT, we can prove the following bound for the probability of choosing the best individual:

$$\begin{aligned} p_b &= \frac{\frac{1}{v+f(b)}}{\sum_{g \in G} \frac{1}{v+f(g)}} \\ &= \frac{\frac{1}{v+\mathcal{A}(b)+\omega(\delta(b))}}{\sum_{g \in G} \frac{1}{v+\mathcal{A}(g)+\omega(\delta(g))}} \\ &\leq \frac{\frac{1}{v+\mathcal{A}(b)}}{\sum_{g \in G} \frac{1}{v+\mathcal{A}(g)+c}} \\ &\leq \frac{1}{v+\mathcal{A}(b)} \cdot \frac{1}{m \cdot \frac{1}{v+\mathcal{A}(w)+c}} \\ &= \frac{1}{m} \cdot \frac{v+\mathcal{A}(w)+c}{v+\mathcal{A}(b)} \ . \end{aligned} \tag{6}$$

The bound in Equation 6 has several consequences. First, the parameter v for the fitness proportional selection (see Equation 3) should have a low value, because $lim_{v \to \infty} p_b = 1/m$. In other words, if v is too high, then there would be no

selection pressure. Second, if $\mathcal{A}(b) > 0$ and $\mathcal{A}(b) = \mathcal{A}(w)$, then:

$$p_b \leq \frac{1}{m} \cdot \frac{v + \mathcal{A}(w) + c}{v + \mathcal{A}(b)} \leq \frac{1}{m} \cdot \frac{\mathcal{A}(b) + c}{\mathcal{A}(b)} \leq \frac{2}{m} \ . \quad (7)$$

This has the effect that *regardless* of the fitness values of the individuals in the population G, the best individual would have at most probability $2/m$ to be chosen. Notice that the condition $\mathcal{A}(b) > 0$ and $\mathcal{A}(b) = \mathcal{A}(w)$ is quite common. It would represent a particularly difficult predicate to satisfy that is not the target branch (which could be for example nested to it) (i.e., $\mathcal{A}(b) > 0$). After some generations, most individuals could be able to reach that difficult branch without solving it (i.e., $\mathcal{A}(b) = \mathcal{A}(w)$).

Let's consider a numerical simulation: $c = 1$, $|G| = 10$, $\omega = \omega_1$, $\delta(b) = 1$ whereas for all the other individuals g we have $\delta(g) = 1000$. In other words, there is one individual that is far much better than all the others. Figure 6 shows the probability p_b when $v = 10^i$ with $i \in \{-5, -4, \ldots, 3, 4\}$ and $\mathcal{A}(b) = \mathcal{A}(w) \in \{1, \ldots, 10\}$. As Equation 7 predicts, in Figure 6 the selection pressure is very low, i.e. $p_b \leq 2/m = 0.2$. In the experiments, it is very close to the lower bound $1/m = 0.1$.

This result is independent of the normalising function ω, and it would be already enough to suggest to not to use fitness proportional selection for a fitness function such as the one in Equation 1. However, because the focus of this work is in comparing ω_1 against ω_0, we hence conclude our analysis by considering the case $\mathcal{A}(b) = \mathcal{A}(w) = 0$. In this case, ω plays an important role.

For function ω_1, we obtain the following bound for p_b^1:

$$\begin{aligned} p_b^1 &= \frac{\frac{1}{v+\omega_1(\delta(b))}}{\sum_{g \in G} \frac{1}{v+\omega_1(\delta(g))}} \\ &= \frac{\frac{1}{v+(\delta(b)/(\delta(b)+\beta))}}{\sum_{g \in G} \frac{1}{v+(\delta(g)/(\delta(g)+\beta))}} \\ &= \frac{\delta(b)+\beta}{(v+1)\delta(b)+v\beta} \cdot \frac{1}{\sum_{g \in G} \frac{1}{v+(\delta(g)/(\delta(g)+\beta))}} \\ &\leq \frac{\delta(b)+\beta}{(v+1)\delta(b)+v\beta} \cdot \frac{(v+1)\delta(w)+v\beta}{m(\delta(w)+\beta)} \\ &= \frac{1}{m} \cdot \frac{(v+1)\delta(w)+v\beta}{(v+1)\delta(b)+v\beta} \cdot \frac{\delta(b)+\beta}{\delta(w)+\beta} \ . \end{aligned} \quad (8)$$

If we consider v as a constant, it is important to show what type of bounds we would obtain with extreme values of β. This would be important to give rules of thumbs as such as to use small or large values for β regardlessly of the SUT. From Equation 8 it follows:

$$\lim_{\beta \to 0} p_b^1 = \lim_{\beta \to \infty} p_b^1 = 1/m \ .$$

However, if we choose $v = 1/\beta$, then we obtain:

$$\lim_{\beta \to \infty} p_b^1 \leq \frac{1}{m} \cdot \frac{\delta(w)+1}{\delta(b)+1} \ .$$

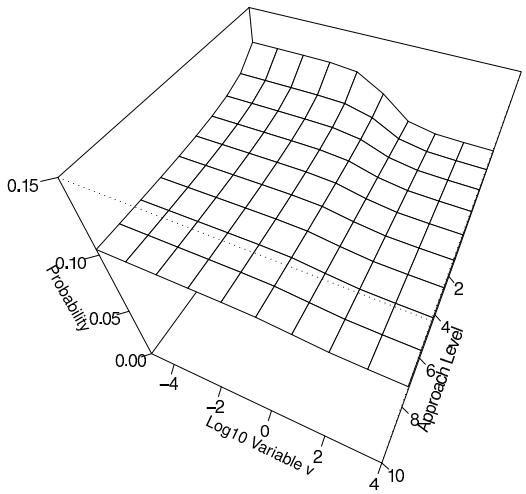

Figure 6. Probability of choosing best individual when $\mathcal{A}(b) > 0$ and $\mathcal{A}(b) = \mathcal{A}(w)$.

In other words, we need to choose the parameter β based on the value v we choose for Equation 3. However, having a high upper bound does not necessarily mean that p_b^1 will have a high value. We hence investigated this issue with a simulation (with the same parameters of the previous simulation). We considered the same values for both v and β, i.e. $\beta = 10^i$ with $i \in \{-5, -4, \ldots, 3, 4\}$. Figure 7 shows that p_b^1 can obtain a high value, but only when β is inversely proportional to v.

For the normalising function ω_0, we obtain the following bound for p_b^0:

$$\begin{aligned} p_b^0 &= \frac{\frac{1}{v+\omega_0(\delta(b))}}{\sum_{g \in G} \frac{1}{v+\omega_0(\delta(g))}} \\ &= \frac{1}{v+1-\alpha^{-\delta(b)}} \cdot \frac{1}{\sum_{g \in G} \frac{1}{v+1-\alpha^{-\delta(g)}}} \\ &\leq \frac{1}{m} \cdot \frac{v+1-\alpha^{-\delta(w)}}{v+1-\alpha^{-\delta(b)}} \ . \end{aligned} \quad (9)$$

Again, if we consider v as a constant, from Equation 9 it follows:

$$\lim_{\alpha \to 1} p_b^0 = \lim_{\alpha \to \infty} p_b^0 = 1/m \ .$$

But this bound does not necessarily converge to $1/m$ if we choose α based on the value v. For example:

$$\lim_{\alpha \to 1, v \to 0} p_b^0 \leq \frac{1}{m} \cdot \frac{\delta(w)}{\delta(b)} \ .$$

Figure 8 shows the results of a similar simulation we did for ω_1 (Figure 7). In this simulation, we use $\alpha = 1 + 10^i$ with $i \in \{-5, -4, \ldots, 3, 4\}$. Similarly to ω_1, also in the case of ω_0 we can obtain a high probability p_b^0. But this happens only when α is properly chosen depending on the value v.

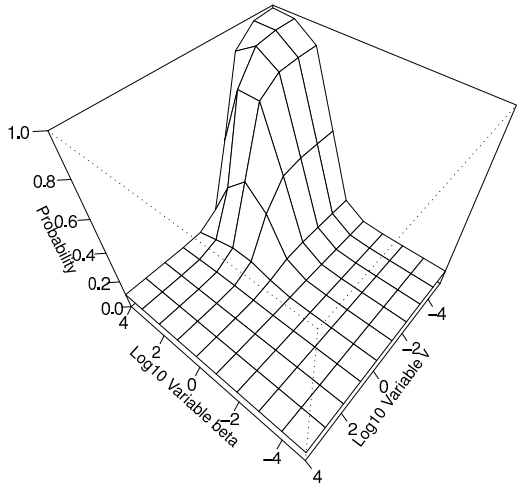

Figure 7. Probability of choosing best individual when $\mathcal{A}(b) = \mathcal{A}(w) = 0$ and $\omega = \omega_1$.

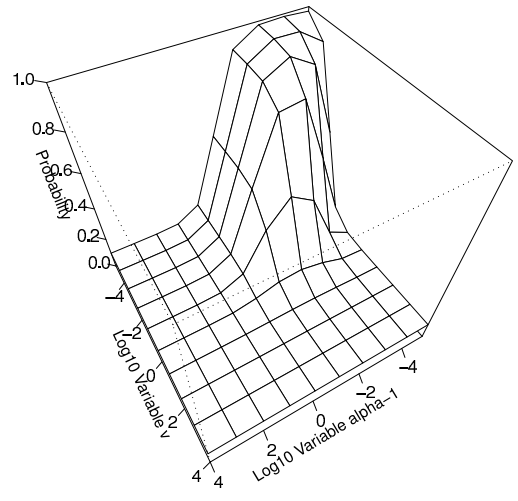

Figure 8. Probability of choosing best individual when $\mathcal{A}(b) = \mathcal{A}(w) = 0$ and $\omega = \omega_0$.

VI. Threats to Validity

Threats to validity of this study are:

- Different results can be obtained for different values of α and β. Their optimal choice is problem dependent.
- We only considered a simple implementation of ω_0 (see Figure 1). We cannot hence exclude that there could be better implementations for ω_0 that lead to prefer it over ω_1.
- In our simulations, only a finite set of values for the considered variables was used. We cannot exclude that very different results could be obtained for larger or smaller values.
- Although our simulations and empirical experiments confirm the analytical analyses, it is possible that all of them contain errors that hide each other.
- For software testing, there could be search algorithms that are better than the two discussed in this paper. For them, it could be possible that the choice of the normalising function would have practically no effect on their runtime. For example, they could be robust to rounding errors in the fitness function.
- On some real-world software, the cost of running test cases could be so high that any optimisation of the code of the normalising function would have no practical effect.
- Rounding errors happen for high values of the branch distance. It could be possible that these high values would seldom be used/found in practice. Unfortunately, this type of information is missing in the literature.

VII. Conclusion

In this paper we have analysed the role of the normalising function for search based software testing. In particular, we focused on branch coverage.

The normalising function has been considered only as a lesser component in the literature. In this paper, we provided theoretical and empirical evidences that actually it can significantly effect the testing process. We studied its role in two commonly used search algorithms, i.e. Simulated Annealing and Genetic Algorithms. Our analytical results are independent of the SUT.

Based on our analyses, we criticise the normalising function that is used in the literature (i.e., ω_0). We hence proposed a new function ω_1 that has lower computational cost and it is more robust to rounding errors. Furthermore, for Simulated Annealing it provides less influence to the current branch distance. We can hence claim that the new proposed ω_1 seems better than ω_0.

In the case of Genetic Algorithms, we found out that there is not much difference in using ω_1 instead of ω_0. However, their parameters α and β need to be properly chosen. We gave theoretical sound advices on how to set β (i.e., $\beta = 1/v$).

Although our focus was on studying the normalising function, during our analyses we found out that fitness proportional selection should not be used in this context regardlessly of the normalising function. This selection mechanism has been already criticised for other reasons (see for example [24]). We provided further arguments against its use that are specific to software testing.

The results of this work are of easy and general application. Most search based testing tools use a normalising function. Applying ω_1 (with $\beta = 1$) would require to change only few lines of code in these tools.

Future work will focus on analysing the role of the normalising function on other common search algorithms, as for example Ant Colony (which has been used for example in [18]). An analysis of the actual code used in the current testing tools will be helpful to understand which other

normalising functions are actually used (this information is unfortunately often missing in the literature). They could be better than ω_1 and its simple implementation we provided.

VIII. Acknowledgments

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the Itea2 VERDE project.

References

[1] A. Arcuri, P. Lehre, and X. Yao. Theoretical runtime analysis in search based software engineering. Technical Report CSR-09-04, University of Birmingham, 2009.

[2] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[3] A. Baresel. Automatisierung von strukturtests mit evolutionŁren algorithmen. Master's thesis, Humboldt-University zu Berlin, 2000.

[4] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[5] A. P. Engelbrecht. *Computational Intelligence: An Introduction*. John Wiley & Sons, 2003.

[6] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Constructing multiple unique input/output sequences using metaheuristic optimisation techniques. *IEE Proceedings - Software*, 152(3):127–140, 2005.

[7] M. Harman and B. F. Jones. Search-based software engineering. *Journal of Information & Software Technology*, 43(14):833–839, 2001.

[8] M. Harman, S. A. Mansouri, and Y. Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, King's College, 2009.

[9] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*. (to appear).

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems, second edition*. MIT Press, Cambridge, 1992.

[11] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[13] N. Mansour, R. Bahsoon, and G. Baradhi. Empirical comparison of regression test selection algorithms. *Journal of Systems and Software*, 57(1):79–90, 2001.

[14] N. Mansour and M. Salame. Data generation for path testing. *Software Quality Control*, 12(2):121–136, 2004.

[15] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[16] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering Methodology*, 18(3), 2008.

[17] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to searchbased test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–24, 2006.

[18] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 2488–2500, 2003.

[19] M. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[21] F. Neumann, P. S. Oliveto, and C. Witt. Theoretical analysis of fitness-proportional selection: landscapes and efficiency. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 835–842, 2009.

[22] H. Waeselynck, P. T. Fosse, and O. A. Kaddour. Simulated annealing applied to test generation: landscape characterization and stopping criteria. *Empirical Software Engineering*, 12(1):35–63, 2006.

[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] D. Whitley. The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89)*, pages 116–121, 1989.

[25] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[26] 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.