Theoretical Runtime Analysis in Search Based Software Engineering

Andrea Arcuri, Per Kristian Lehre and Xin Yao

The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK. Email: a.arcuri,p.k.lehre,x.yao@cs.bham.ac.uk

Abstract

Search algorithms have been used to tackle software engineering problems with promising results. Although the field has attracted a lot of attention recently, it still lacks a theoretical foundation. It has been empirically shown that search algorithms are successful in some software engineering tasks, but we need to understand *why* and *when* they are successful. The long term goal is to get insight of how search algorithms work on software engineering problems, so we can exploit this knowledge to design more efficient algorithms. *Runtime Analysis* is a type of theoretical investigation that aims to determine, via rigorous mathematical proofs, the *time* a search algorithm needs to find an optimal solution. Runtime analysis has previously been carried out on traditional combinatorial problems. In this paper, we advocate that runtime analysis would be helpful in search based software engineering as well. We give the first runtime analysis of search heuristics in the software testing domain.

Keyword: Runtime Analysis, Search Algorithms, Search Based Software Engineering, Software Testing.

1 Introduction

Many tasks in software engineering are extremely expensive. For example, it is very common that software testing can take up to half the resources of the development of a new software [5]. This is why a large part of the literature is devoted to automating as many of these tasks as possible.

Several different techniques have been developed and applied to real world software engineering problems. Among them, there is the use of search algorithms (e.g., Genetic Algorithms [24]). They have been very successful in many engineering domains, and their promising use in software engineering has led to what is now called *Search Based Software Engineering* (SBSE) [21, 20].

Although there has been a lot of research on SBSE in recent years (e.g, in software testing [35]), there exist few theoretical results. The only exceptions we are aware of are on computing unique input/output sequences for finite state machines [32, 31], the application of the Royal Road theory to evolutionary testing [22], and the first versions of our work [2, 1].

To get a deeper understanding of the potential and limitations of the application of search algorithms in software engineering, it is essential to complement the existing experimental research with theoretical investigations. *Runtime Analysis* is an important part of this theoretical investigation, and brings the evaluation of search algorithms closer to how algorithms are classically evaluated.

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

The field of runtime analysis has now advanced to a point where the runtime of relatively complex search algorithms can be analysed on classical combinatorial optimisation problems [44]. We advocate that this type of analysis in SBSE will be helpful to get insight on how search algorithms behave in the software engineering domain. The final aim in the *long term* is to exploit the gained knowledge to design more efficient algorithms.

In this paper we review runtime analysis and we explain how it can be applied to SBSE. We start our analysis on software testing because this is the most studied sub-field of SBSE. In particular, we focus on *branch coverage* in White Box Testing [39].

There can be at least two main directions of research:

- Runtime can be studied on different types of predicates, relations among the input variables, different structures of the control flow graph, etc. The resulting theorems would hence be used as basic *building blocks* to calculate the runtime of the classes of software that can be built with these blocks. The applicability of the results would be very wide, because we would have precise runtime for an infinite number of software. This type of analysis would help to understand which are the properties of software that make hard the search for test data. Unfortunately, proving that a software has some particular properties (for which we could have precise runtimes) would be hard in general.

- Runtime can be theoretically calculated on specific software that are commonly used in literature, like for example the *Space* program [9, 36] and Java containers [53, 3]. Because they are widely used, it would be helpful to get stronger theoretical results about them. A better understanding of *how* search algorithms behave on these problems would help to make more precise and rigorous comparisons in empirical validations of novel techniques against common search algorithms. Theorems on specific software would not be applicable to other case studies. However, there is similar issue of generalisation in empirical studies, because behaviour of search algorithms is strongly dependent on the tackled instances of the problem. Empirical studies are more easy to carry out than theoretical analysis, hence larger case studies would lead to more generalisable results. But once a precise theoretical analysis is given for a testing problem, that would be a rigorous and exact result that can be reused each time that testing problem is used in an empirical study.

In this paper we focus on the second direction of research. In fact, we believe that for the first step it is more appropriate to get theoretically results on well known testing problems.

We study the runtime of five different search algorithms on test data generation for branch coverage of the *Triangle Classification* (TC) problem [39]. We chose TC because it is the *most famous* problem in software testing and it is amenable to rigorous mathematical treatment without being distracted by too many details. The search algorithms considered for the analyses are: Random Search (RS), Hill Climbing (HC), Alternating Variable Method (AVM), (1+1) Evolutionary Algorithm (EA), and (μ+1) Steady State Genetic Algorithm (SSGA).

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 separate search for each branch. However, analyses on the dependency 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. In some frameworks, the solutions found so far are exploited (e.g., smart seeding strategies) to guide the search of the remaining uncovered branches. However, for the sake of simplicity, we consider each branch as an independent search problem. Because there is a constant number of branches, the asymptotic runtime of a search algorithm is determined only by the most expensive branch.

We start our analysis with an empirical study on each branch of TC. We then carry out a theoretical study for RS on each of the 12 branches. On the branch that seems the most difficult to cover (branch ID_8, see Figure 1 in Section 3), we make a theoretical runtime analysis of HC, AVM and (1+1) EA. The analysis shows that AVM has the lowest runtime on this branch.

Any empirical analysis on randomised algorithms is subject to stochastic variations. Furthermore, the branches that are easy for RS may not be necessarily easy for other search algorithms. Therefore, we make a theoretical study of AVM also on all the other 11 branches to confirm that it is actually the fastest. In fact, the runtime of AVM on the branch for which it has maximal runtime, is lower than the maximal runtime of any of the other analysed search algorithms.

For HC and (1+1) EA, we also theoretically analyse their runtime on a simple branch (i.e., branch ID_0). For each considered search algorithm, for branch ID_8 we theoretically study different fitness functions, and what is the expected number of steps that these algorithms make at most.

The main contributions of this paper are:

- As far as we know, this is the first extensive work on runtime analyses of search algorithms applied to search based software testing. Although the presented theorems are specific to a particular case study, the methodology to obtain these results can be used in general to other SBSE problems.
- TC is the most famous and used case study in the literature of software testing. We provide a rigorous theoretical analysis of this testing problem. The obtained results can be used in any future empirical study in which this case study is employed. For example, if a newly designed algorithm empirically seems faster than AVM on TC, then there is no need to compare it against RS, HC and (1+1) EA on TC.
- Often the TC problem is used as an example to show the limitations of RS and hence to validate the study of more complex search algorithms. However, we have proved that RS is not the worst on at least one of the branches of TC.
- We prove that there exists at least one search algorithm (i.e., AVM) that has a runtime complexity that is strictly better than that of RS on at least one testing problem (i.e, TC).
- We prove that some search algorithms have a high probability of finding an optimal solution in reasonable time (a more precise description is given in the following sections).
- Algorithms that seem to perform poorly, in comparison with others, might perform much better when the size of the problem increases (i.e., they might scale up better). We prove that this is in fact the case for a non-trivial case in the software testing domain.

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 the five different search algorithms applied to find test data for TC. Empirical studies follow in Section 5, whereas theoretical analyses are presented in Section 6. The obtained results and their implications are discussed in Section 7. Finally, Section 8 concludes the paper.

2 Runtime Analysis

Evolutionary algorithms and other randomised search heuristics are attractive due to their versatility. However, in contrast to many problem specific algorithms, it can be notoriously difficult to establish exactly how these algorithms work, and why they sometimes fail. Empirical investigations can be costly and do not always yield the desired level of information needed to make the right choice of heuristic and corresponding parameter setting at hand.

To put the application of search heuristics in software engineering and other domains on a firmer ground, it is desirable to construct a theory which can explain the basic principles of the heuristics and possibly provide guidelines for developing new and improved algorithms. Such a theory should preferably be valid without making simplifying assumptions about the algorithms or problems, e.g. assuming that the EA has infinite population size, or ignoring the stochastic nature of these algorithms.

When studying a particular search heuristic, it is important that one makes clear what class of problems one has in mind. One can say very little about the advantages and disadvantages of a heuristic without making any assumption about the problem [60].

For a given heuristic and problem class, an initial theoretical question to ask, is whether the heuristic will ever find a solution, if it is allowed unlimited time. This type of questions falls within the realms of convergence analysis, which is a well-developed area [49]. There exist simple conditions on the underlying Markov chain of a search heuristic that guarantee convergence in finite time. These conditions often hold for the popular heuristics [49]. Note that convergence itself gives very little information about whether an algorithm is practically useful, because no limits are put on the amount of resources the algorithm uses.

In this paper, we are concerned with the harder question of determining how long the heuristic needs to find the solution. In line with the analysis of classical algorithms [6], we will seek to find a relationship between the size of a problem and the number of basic steps needed to find the solution.

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* [13], in which time is measured as the number of times the algorithm evaluates the cost function.

Definition 1 (Runtime [12, 23]). *Given a class $\mathcal{F}$ of cost functions $f_i : S_i \to \mathbb{R}$, the* runtime $T_{A,\mathcal{F}}(n)$ *of a search algorithm A is defined as*

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

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

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

The last decades of research in the area show that it is important to apply appropriate mathematical techniques to get good results [55]. Initial studies of exact Markov chain models of search heuristics were not fruitful, except for the the simplest cases.

A more successful and particularly versatile technique has been so-called drift analysis [23, 45], where one introduces a potential function which measures the distance from any search point to the global optimum. By estimating the expected one-step drift towards the optimum with respect to the potential function, one can deduce expected runtime and success probability. Finding the right potential function can sometimes be a challenge, and, as in the case of Definition 9 in Section 6.6, can be considerably different from the objective function.

In addition to drift analysis, the wide range of techniques used in the study of randomised algorithms [38], in particular Chernoff bounds, have proved useful also for evolutionary algorithms.

Initial studies of runtime were concerned with simple EAs like the (1+1) EA on artificial pseudo-boolean functions [11, 12, 56]. These studies established fundamental facts about the (1+1) EA, e.g.

that it can optimise any linear function in $O(n \log n)$ expected time [12], that quadratic functions with negative weights are hard [56], that the hardest functions require $\Theta(n^n)$ iterations [12] and, in contrast to commonly held belief, that not all unimodal functions are easy [11].

The understanding of the runtime of search heuristics were expanded in several directions, by analysing more complex algorithms, by considering a wider range of problems, and by considering different problem settings, e.g. multi-objective optimisation [15, 40], co-evolutionary optimisation and optimisation in continuous domains [25].

Runtime analysis on artificial functions has provided a better understanding of fundamental aspects of EAs, e.g. under which conditions algorithmic parameters play a particularly important role, e.g. the crossover operator [27, 51], populations in single [59] and multi-objective optimisation [16], and diversity mechanisms [14]. Furthermore, the analysis of a wide range of search heuristics, including ant colony optimisation [42] and particle swarm optimisation [52] has been initiated on pseudo-boolean functions.

The analysis of search heuristics expanded to classical combinatorial optimisation problems, and many of these results are covered in the survey [44]. Initially, combinatorial optimisation problems in $\mathcal{P}$ were analysed [17, 50, 41, 10]. Giel *et al* showed that although the runtime of (1+1) EA is in general exponential, the EA is a polynomial-time randomised approximation scheme (PRAS) for the problem [17]. Other problems analysed include sorting [50], minimum spanning tree [41] and Eulerian cycle [10].

It is unrealistic to hope that the expected runtime of any search heuristics on the worst case instances of $\mathcal{NP}$-hard problems is anything less than exponential. Instead, one can focus on analysing the runtime on interesting sub-classes of the problem, e.g. the vertex cover problem [43], on the average case runtime over the set of instances, or the approximation quality that can be obtained by the algorithm in polynomial time. There exists relatively few results in this area, however it is worth noting the average case analysis by Witt of (1+1) EA on the partition problem [58].

Runtime analysis is practically unexplored within search based software engineering. This might be due to the fact that many of these problems have been outside the reach of the analysis techniques available, partly because many software engineering problems are $\mathcal{NP}$-hard [20].

Lehre and Yao considered conformance testing of finite state machines, and analysed the runtime of (1+1) EA on the problem of computing unique input output sequences [32]. Theoretical runtime results confirmed existing experimental results [19],[8] that EAs can outperform random search on the UIO problem, showing that the expected running time of (1+1) EA on a counting FSM instance class is $O(n \log n)$, while random search needs exponential time [32]. The UIO problem is $\mathcal{NP}$-hard [29], so one can expect that there exist EA-hard instance classes. It has been proved that a combination lock FSM is hard for the (1+1) EA [32]. To reliably apply EAs to the UIO problem, it is necessary to distinguish easy from hard instances. Theoretical results indicate that there is no sharp boundary between these categories in terms of runtime. For any polynomial n^k, there exist UIO instance classes where the (1+1) EA has running time $\Theta(n^k)$ [33].

Recent work has investigated the impact on runtime of the acceptance criterion in (1+1) EA and the crossover operator in (μ+1) SSGA when computing UIOs from FSMs [31]. The results show some instance classes where the right choice of acceptance criterion is essential. Furthermore, the results point out cases where crossover and a large population are essential for (μ+1) SSGA to compute the UIO in polynomial time [31].

3 Triangle Classification Problem

TC is the most famous problem in software testing. It opens the classic 1979 book of Myers [39], and has been used and studied since early 70s (e.g., [18, 48, 7]). However, the true origin of TC is not completely clear [61]. At any rate, TC is still widely used in many publications (e.g., [34, 57, 35, 30, 37, 62]).

We use the implementation for the TC problem that was published in the survey by McMinn [35] (see Figure 1). Some slight modifications to the program have been introduced for clarity.

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

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

Although limits on the input variables are common in software testing, there is usually no guarantee that there exists a global optimum within those limits. However, the increase in runtime for a given increase in variable range, gives useful information. For example, what are the consequences of choosing a too wide range?

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

In our analyses, the size n of the problem defines the range $R = \{-n/2 + 1, \ldots, n/2\}$ in which the variables in I can be chosen (i.e., $x, y, z \in R$). Hence, the search space S is defined as $S = \{(x, y, z)|x, y, z \in R\}$, and it is composed of n^3 elements. Without loss of generality n is even and multiple of 4. To obtain full coverage, it is necessary that $n \geq 8$, otherwise the branch regarding the classification as *scalene* will never be covered. Note that one can consider different types of R (e.g., $R' = \{0, \ldots, n\}$), and each type may lead to different behaviours of the search algorithms. We based our choice on what is commonly used in literature. For simplicity and without loss of generality, search algorithms are allowed to generate solutions outside S. In fact, R is mainly used when random solutions need to be initialised.

The search space is composed of 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 clearer.

The employed fitness function f is the commonly used approach level $\mathcal{A}$ plus the branch distance δ [35]. For a target branch z, we have that the fitness function f_z is:

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

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

```
int tri_type(int x, int y, int z) {
  int type;
  int a=x, b=y, c=z;
  if (x > y) { /* ID_0 */
    int t = a; a = b; b = t;
  } else { /* ID_1 */}
  if (a > z) { /* ID_2 */
     int t = a; a = c; c = t;
  } else { /* ID_3 */}
  if (b > c) { /* ID_4 */
     int t = b; b = c; c = t;
  } else { /* ID_5 */}
  if (a + b <= c) { /* ID_6 */
     type = NOT_A_TRIANGLE;
  } else { /* ID_7 */
      type = SCALENE;
      if (a == b && b == c) {
          /* ID_8 */
          type = EQUILATERAL;
      } else /* ID_9 */
             if (a == b || b == c) {
                 /* ID_10 */
                 type = ISOSCELES;
             } else {/* ID_11 */}
    }
  return type;
}
```

Figure 1: Triangle Classification (TC) program, adapted from [35]. Each branch is tagged with a unique ID.

to guarantee it is to have $\mathcal{A}_z(I_{i+1}) = \mathcal{A}_z(I_i) + \zeta$, where ζ can be any positive constant (e.g., $\zeta = 1$) and $\mathcal{A}_z(I_0) = 0$.

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

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 ω does not have any effect besides its computational cost. An example, for which this would not apply, is the use of "Fitness Proportional Selection" in GAs.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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. The general versions, that consider this length as a problem parameter, would have the same computational behaviour in terms 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. Therefore the termination criterion is left unspecified to simplify the description of the algorithms.

The fitness function depends on the target branch. We used the common function used in search based software testing [35], i.e. the sum of the approach level with the branch distance. We also carried out analyses with only the approach level without the branch distance.

4.1 Random Search

RS is the simplest search algorithm. It 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 about previously visited points when choosing the next point to sample. Often, RS is used as a baseline for evaluating the performance of other more sophisticated meta-heuristics.

It is important not to confuse RS in white box testing with Random Testing. In random testing, 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.

Because RS does not exploit any gradient in the objective function, there is no difference in using the branch distance or not in the fitness function.

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

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

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 I'. The same procedure of looking at the neighbour solutions is then repeated on I_1, until a final point I_i is reached, where $\forall I' \in N(I_i) : f(I') \geq f(I_i)$, 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.

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 ψ 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 for solution I_i is the set of solutions:

$$N_d(I_i) := \{I_i + d \mid d \in D \text{ and } I_i + d \in S\} ,$$

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 ψ, we do not need to define one. In fact, the following theoretical 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. For the empirical study, we chose a simple strategy ψ that moves each time to the first better solution it finds.

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 $\psi$
        if $f(I') < f(I)$, then
            $I := I'$.
```

4.3 Alternating Variable Method

AVM is similar to a HC, and was employed in the early work of Korel [28]. The algorithm starts on a random search point I, and then it considers modifications of the input variables, one at a time. The algorithm applies an *exploratory search* to the chosen variable, in which the variable is slightly modified (in our case, by ± 1). If one of the neighbours has a better fitness, then the exploratory search is considered successful. Similarly to HC, the better neighbour will be selected as the new current solution. Moreover, a *pattern search* will take place. On the other hand, if none of the neighbours has better fitness, then AVM continues to do exploratory searches on the other variables, until either a better neighbour has been found or all the variables have been unsuccessfully explored. In this latter case, a restart from a new random point is done if a global optimum was not found.

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

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

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

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

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

4.4 (1+1) Evolutionary Algorithm

Runtime analysis of evolutionary algorithms is difficult and only recently have rigorous results become available. When initiating the analysis in a new problem domain, it is an important first step to analyse a simple algorithm like the (1+1) EA. Without understanding the behaviour of such a simple algorithm in the new domain, it is difficult to understand the behaviour of more complex EAs, e.g. those EAs that use a population and crossover. Although the (1+1) EA is relatively simple compared to other evolutionary algorithms, recent research has shown that this algorithm is surprisingly efficient on a wide range of useful problems [44], including sorting [50], minimum spanning tree [41] and Eulerian cycle [10].

Definition 7 ((1+1) EA)**.**

```
*Choose* $x$ *uniformly from* $\{0,1\}^n$.
**Repeat**
    $x' := x$.
    *Flip each bit of* $x'$ *with probability* $1/n$.
    **If** $f(x') \geq f(x)$,
        **then** $x := x'$.
```

4.5 Genetic Algorithms

The most used search algorithms in the literature of SBSE are the Genetic Algorithms (GAs) [24].

GAs are a global search meta heuristic inspired by the Darwinian Evolution theory. Different variants of this meta heuristic exist. However, 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 offsprings 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.

In our analyses, we used a simple steady state implementation (SSGA).

Definition 8 ((μ+1) Steady State Genetic Algorithm (SSGA))**.**

```
*Sample a population* $P$ *of* $\mu$ *points u.a.r. from* $S$.
**repeat**
    **with probability** $p_c(n)$,
        *Sample* $x$ *and* $y$ *u.a.r. from* $P$.
        $(x', y') :=$ *one point crossover*$(x, y)$.
        **if** $\max\{f(x'), f(y')\} \geq \max\{f(x), f(y)\}$
            **then** $x := x'$ *and* $y := y'$.
    **otherwise**
        *Sample* $x$ *u.a.r. from* $P$.
        *Flip each bit of* $x'$ *with probability* $1/\ell(x')$.
        **if** $f(x') \geq f(x)$
            **then** $x := x'$.
```

5 Empirical Study

Comparing theoretical analyses against empirical ones is useful to see which different types of information they can give.

Table 1: Models used for the non-linear regression. The constant ρ is the model parameter that is estimated with the regression.

Runtime models
$\rho \cdot 1$
$\rho \cdot \log(n)$
$\rho \cdot \log(n)^2$
$\rho \cdot n$
$\rho \cdot n \log(n)$
$\rho \cdot n \log(n)^2$
$\rho \cdot n^2$
$\rho \cdot n^2 \log(n)$
$\rho \cdot n^2 \log(n)^2$
$\rho \cdot n^3$

We ran each search algorithm on each branch of TC for the following values of n: $\{16, 32, 64, 128, 256, 512, 1024\}$. For each size of n, we ran 30 trials (with different random seeds) and recorded the number of fitness evaluations done before reaching a global optimum. We used the fitness functions in Figure 2. We also ran experiments with only the approach level, i.e. without using the branch distance.

Following [26], for each setting of algorithm and problem instance size, we fitted different models to the observed runtimes using non-linear regression with the Gauss-Newton algorithm. Each model corresponds to a one term expression $\rho \cdot t(n)$ of the runtime, where the model parameter ρ corresponds to the constant to be estimated. The residual sum of squares of each fitted model was calculated to identify the model which corresponds best with the observed runtimes. This methodology was implemented in the statistical tool R [47]. Ten different runtime models were considered (shown in Table 1). Note that the ten models were chosen before the theoretical investigation was started, and that choice was made based on what we thought would be appropriate.

The models with lowest error are shown in Table 2. The branch that seems most difficult to cover is the one related to the classification as *equilateral*, i.e. ID_8.

To study the effect that the size of data has on the accuracy of the models, we carried out another set of experiments. We used a size set $S = \{16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$, and we run experiments with different ordered subsets of S (8 in total), as for example $\{16, 32, 64\}$, ..., $\{16, \ldots, 8192\}$. For each size in S we run 30 trials. Table 3 shows the results for branch ID_8. The fitness function uses the branch distance.

At any rate, this type of empirical analysis has the following limitations:

- although more experimental data could lead to infer the right models, it is *a priori* difficult to estimate how much data is needed for obtaining them. Moreover, experiments might be computationally expensive, hence it might be not possible to obtain the right amount of data.

- if an algorithm has an high complexity (e.g., $\Theta(2^n)$ or $\Theta(n^5)$), only experiments with low values of n can be carried out. That limits the accuracy of the model (e.g., there might be not much difference between n^5 and $n^4 \log(n)^2$).

- when we try to fit a set of models, the correct one might not be necessarily among them.

Table 2: Result of the empirical study. Each branch has an ID based on its order in the code. For each branch, there are shown the results whether the branch distance was used (T) or not (F).

Branch ID	Branch Distance	RS	HC	AVM	(1+1) EA	(μ+1) SSGA
ID_0	T	2.2000	0.9497 $\log(n)^2$	0.4558 $\log(n)$	1.1470 $\log(n)$	2.0480
	F	2.2380	7.8480	8.0810	1.3270 $\log(n)$	2.0520
ID_1	T	1.9570	0.0173 $n\log(n)$	0.5796 $\log(n)$	5.5710	1.9050
	F	1.8190	7.6380	0.9658 $\log(n)$	1.0750 $\log(n)$	1.9570
ID_2	T	3.2140	0.0008 n^2	1.1690 $\log(n)$	10.6100	3.3430
	F	3.2480	15.0100	12.6400	2.0560 $\log(n)$	3.3950
ID_3	T	1.4100	0.1435 n	0.4358 $\log(n)$	3.5670	1.5330
	F	1.5570	4.5140	0.5952 $\log(n)$	0.5458 $\log(n)$	1.4330
ID_4	T	1.4860	0.0018 $n\log(n)^2$	0.4037 $\log(n)$	0.5677 $\log(n)$	1.5240
	F	1.4430	4.2190	4.7520	3.8190	1.5330
ID_5	T	0.3781 $\log(n)$	0.0315 $n\log(n)$	0.9759 $\log(n)$	12.3300	3.1140
	F	2.7480	14.1900	1.6010 $\log(n)$	1.6730 $\log(n)$	2.6670
ID_6	T	1.0710	0.0126 n	1.2430	1.1240	1.0480
	F	1.0810	1.2670	1.2430	0.1790 $\log(n)$	1.0670
ID_7	T	15.4000	0.9594 n	4.4670 $\log(n)$	2.1370 $\log(n)^2$	30.5600
	F	17.7400	95.5200	98.7700	10.6500 $\log(n)$	48.9400
ID_8	T	0.2476 $n^2\log(n)$	1.3670 n	6.8240 $\log(n)$	5364.0000 n	37.3000 $\log(n)^2$
	F	2.4110 n^2	0.0235 $n^2\log(n)^2$	1.7690 n^2	0.0319 $n^2\log(n)$	0.4475 n^2
ID_9	T	15.8300	0.1025 $n\log(n)$	4.8760 $\log(n)$	13.5100 $\log(n)$	37.2200
	F	14.2300	91.7200	93.7400	70.4400	40.9000
ID_{10}	T	1.7310 n	1.1090 n	4.8320 $\log(n)$	0.4556 n	5.6970 $\log(n)^2$
	F	0.2525 $n\log(n)$	3.8490 n	4.5880 n	3.4860 $\log(n)^2$	1.0800 n
ID_{11}	T	17.5900	2.8810 n	31.8500	14.4700 $\log(n)$	41.4700
	F	19.1600	110.8000	105.6000	79.3500	45.8000

Table 3: Result of experiments for branch ID_8. Data were collected with different values of n.

Max n	RS	HC	AVM	(1+1) EA	(μ+1) SSGA
16	1.8970 $n\log(n)^2$	1.8390 n	7.4300 $\log(n)$	0.1839 $n^2\log(n)$	3.9180 $n\log(n)$
32	1.8970 $n\log(n)^2$	1.8390 n	7.4300 $\log(n)$	0.1839 $n^2\log(n)$	3.9180 $n\log(n)$
64	0.0696 $n^2\log(n)^2$	1.6050 n	33.7400	0.1478 n^3	21.3400 $\log(n)^2$
128	2.5960 n^2	1.6400 n	6.5310 $\log(n)$	0.2961 $n^2\log(n)^2$	24.5100 $\log(n)^2$
256	2.2170 n^2	1.6680 n	6.2120 $\log(n)$	14.1200 n^2	10.3500 n
512	2.0150 n^2	1.5510 n	6.3620 $\log(n)$	57.4300 $n\log(n)^2$	35.0800 $\log(n)^2$
1024	1.9560 n^2	1.5240 n	6.4170 $\log(n)$	8.2890 n^2	40.7100 $\log(n)^2$
2048	0.0218 $n^2\log(n)^2$	0.1559 $n\log(n)$	6.6250 $\log(n)$	682.0000 $n\log(n)$	4.2890 n
4096	2.1580 n^2	1.6950 n	6.9210 $\log(n)$	6025.0000 n	0.0025 n^2
8192	2.2910 n^2	1.6920 n	7.2870 $\log(n)$	6279.0000 n	116.7000 $\log(n)^2$

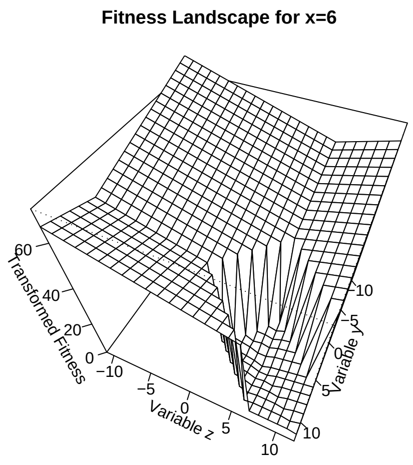

Figure 3: Fitness landscape of modified fitness function f_8 with $n = 24$ and x fixed to the value 6.

6 Theoretical Analysis

For RS and AVM we studied the runtime on each target branch. For HM and (1+1) EA, we only focused on branch ID_8. We have not formally analysed the runtime of SSGA, although we carried out empirical experiments for sake of comparison.

For fitness function f_8, in Figures 3 and 4 we show 3D graphs of the fitness landscape with variable x fixed to $n/4$ and $-n/4$ respectively. The value of n is 24. Because the use of a normalising function ω would make difficult to visualise the difference between the fitness values of the solutions, we do not use a normalising function. Instead, to draw the fitness landscape of f_8 we use $\omega(h) = h$, and then we choose $\zeta = 2n$ high enough to guarantee that higher approach levels give worse fitness values. The value of γ is 1. Note that in Figure 3 there are small plateaus that correspond to the cases when $a + b > c$ and the value of b is modified.

For branch ID_8, when the branch distance is not used, the fitness function is:

$$f(I) = \begin{cases} 2\zeta & \text{if } a + b \leq c\ , \\ 0 & \text{if } a = b \wedge b = c \wedge a + b > c, \text{ and} \\ 1\zeta & \text{otherwise}\ . \end{cases} \qquad (1)$$

6.1 Global Optima

For each branch ID_i, we calculate the number of global optima.

Proposition 1. *For the objective function f_0, considering the space of solutions S, there are $g_0 = (1/2)(n-1)n^2$ global optima.*

Proof. We need to consider all the cases in which $x > y$ and z can assume any value.

$$g_0 = n \sum_{i=1}^{(n-1)} (n-i) = n\big(n(n-1) - (1/2)(n-1)(n)\big) = (1/2)(n-1)n^2\ .$$

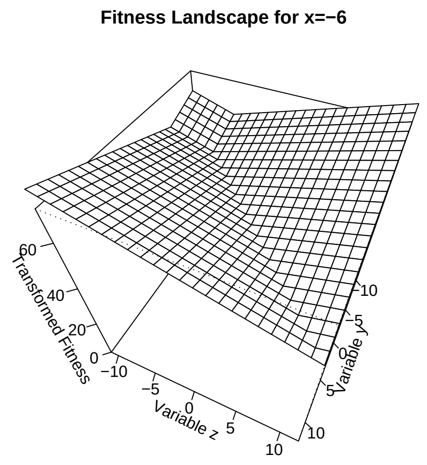

Figure 4: Fitness landscape of modified fitness function f_8 with $n = 24$ and x fixed to the value -6.

$\square$

Proposition 2. *For the objective function f_1, considering the space of solutions S, there are $g_1 = (1/2)(n+1)n^2$ global optima.*

Proof. If branch ID_0 is not executed, then ID_1 is executed. Therefore, using Proposition 1:

$$g_1 = n^3 - (1/2)(n-1)n^2 = (1/2)(n+1)n^2 \ .$$

$\square$

Proposition 3. *For the objective function f_2, considering the space of solutions S, there are $g_2 = (1/6)(n-1)(n)(2n-1)$ global optima.*

Proof. We need to consider all the cases in which $x > z$ and $y > z$.

$$g_2 = \sum_{i=1}^{n-1} (n-i)^2 = (1/6)(n-1)(n)(2n-1) \ .$$

$\square$

Proposition 4. *For the objective function f_3, considering the space of solutions S, there are $g_3 = (1/6)(n)(n+1)(4n-1)$ global optima.*

Proof. If branch ID_2 is not executed, then ID_3 is executed. Therefore, using Proposition 3:

$$g_3 = n^3 - (1/6)(n-1)(n)(2n-1) = (1/6)(n)(n+1)(4n-1) \ .$$

$\square$

Proposition 5. *For the objective function f_4, considering the space of solutions S, there are $g_4 = (1/3)(n)(n-1)(2n-1)$ global optima.*

Proof. We need to consider all the cases in which $\max(x,y) > \max(z, min(x,y))$. There is no valid solution to this inequality if $x = y$. Because $\max(x,y) \geq min(x,y)$, those cases can be simplified to $\max(x,y) > z$ with $x \neq y$.

$$g_4 = 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(j-1) = 2\sum_{i=1}^{n-1}\big((1/2)(n)(n-1) - (1/2)(i)(i-1)\big) = (1/3)(n)(n-1)(2n-1)\ .$$

□

Proposition 6. *For the objective function f_5, considering the space of solutions S, there are $g_5 = (1/3)(n)(n^2+3n-1)$ global optima.*

Proof. If branch ID_4 is not executed, then ID_5 is executed. Therefore, using Proposition 5:

$$g_5 = n^3 - (1/3)(n)(n-1)(2n-1) = (1/3)(n)(n^2+3n-1)\ .$$

□

Proposition 7. *For the objective function f_8, considering the space of solutions S, there are $n/2$ global optima, and they are of the form $G = (t,t,t)$, with $t > 0$.*

Proof. This can be proved by considering fitness function f_8, 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 $R = \{-n/2+1, \ldots, n/2\}$, there are $g_8 = n/2$ possible different t with $t > 0$. □

Proposition 8. *For the objective function f_{10}, considering the space of solutions S, there are $g_{10} = (3/16)(n)(3n-8)$ global optima.*

Proof. We first analyse the case $b = c$.

$$k = \binom{3}{1}\sum_{a=1}^{n/2}((n/2)-a) = (3/2)(n)\sum_{a=1}^{n/2}1 - 3\sum_{a=1}^{n/2}a = (3/8)(n)(n-2)\ .$$

We then need to consider the cases in which $a = b$.

$$t = \binom{3}{1}\Big(\sum_{a=1}^{n/2}\sum_{c=a+1}^{min(2a-1,n/2)}1\Big) = 3\Big(\sum_{a=1}^{n/4}\sum_{c=a+1}^{2a-1}1\Big) + 3\Big(\sum_{a=1+n/4}^{n/2}\sum_{c=a+1}^{n/2}1\Big) = (3/16)(n^2-4n)\ .$$

Finally:

$$g_{10} = k + t = (3/8)(n)(n-2) + (3/16)(n^2-4n) = (3/16)(n)(3n-8)\ .$$

□

Proposition 9. *For the objective function f_{11}, considering the space of solutions S, there are $g_{11} = (1/16)(n)(n-4)(n-5)$ global optima.*

Proof. We need to consider all the cases in which $a \neq b \neq c$ and $a + b > c$. To make this latter predicate true we need $a \geq 2$.

$$\begin{array}{rcl} g_{11} & = & 3!\Big(\sum_{a=2}^{(n/2)-2} \sum_{b=a+1}^{(n/2)-1} \sum_{c=b+1}^{min(a+b-1,n/2)} 1\Big) \\ & = & 6\Big(\sum_{a=2}^{(n/4)-1} \sum_{b=a+1}^{(n/2)-a} \sum_{c=b+1}^{a+b-1} 1\Big) + 6\Big(\sum_{a=2}^{n/4} \sum_{b=(n/2)-a+1}^{(n/2)-1} (n/2) - b\Big) + \\ & & +6\Big(\sum_{a=(n/4)+1}^{(n/2)-2} \sum_{b=a+1}^{(n/2)-1} (n/2) - b\Big) \\ & = & (1/32)(n)(n-4)(n-8) + (1/64)(n)(n+4)(n-4) + (1/64)(n)(n^2-12n+32) \\ & = & (1/16)(n)(n-4)(n-5)\ . \end{array}$$

□

Proposition 10. *For the objective function f_9, considering the space of solutions S, there are $g_9 = (1/16)(n)(n+2)(n-2)$global optima.*

Proof. Using Propositions 8 and 9, we just need to add the global optima for branches ID_{10} and ID_{11}.

$$g_9 = (3/16)(n)(3n-8) + (1/16)(n)(n-4)(n-5) = (1/16)(n)(n+2)(n-2)\ .$$

□

Proposition 11. *For the objective function f_7, considering the space of solutions S, there are $g_7 = (1/16)(n)(n^2+4)$ global optima.*

Proof. Using Propositions 7 and 10, we just need to add the global optima for branches ID_8 and ID_9.

$$g_9 = (n/2) + (1/16)(n)(n+2)(n-2) = (1/16)(n)(n^2+4)\ .$$

□

Proposition 12. *For the objective function f_6, considering the space of solutions S, there are $g_6 = (1/16)(n)(15n^2-4)$ global optima.*

Proof. If branch ID_7 is not executed, then ID_6 is executed. Therefore, using Proposition 11:

$$g_6 = n^3 - (1/16)(n)(n^2+4) = (1/16)(n)(15n^2-4)\ .$$

□

6.2 General Properties

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

Proposition 13. *For the objective function f_8, let $a \leq b \leq c$, and $v > 0$, then $f_8(a,b,c) < f_8(a-v,b,c)$.*

Proof. In the case when $a + b \leq c$, then $a - v + b \leq c$ and we have $f_8(a,b,c) = \zeta + \omega(c - a - b + \gamma)$ and $f_8(a-v,b,c) = \zeta + \omega(c - (a-v) - b + \gamma)$, in which case the proposition holds. Assume on the other hand that $a + b > c$. Let g be:

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

we get:

$$\begin{aligned} f_8(a-v,b,c) & \geq \omega(c - a + v + g) \\ & > \omega(c - a + g) \\ & = f_8(a,b,c)\ . \end{aligned}$$

□

Proposition 14. *For objective function f_8, if $a \leq b \leq c$, and $v > 0$, then $f_8(a,b,c) < f_8(a,b,c+v)$.*

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

$$\begin{aligned} f_8(a,b,c+v) &= \zeta + \omega(v + c - a - b + \gamma) \\ &> \zeta + \omega(c - a - b + \gamma) \\ &= f_8(a,b,c) \ . \end{aligned}$$

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

$$\begin{aligned} f_8(a,b,c+v) &\geq \omega(v + c - a + g) \\ &> \omega(c - a + g) \\ &= f_8(a,b,c) \ , \end{aligned}$$

where g is as defined in the proof of Proposition 13. □

Proposition 15. *Given x and y uniformly and independently distributed in R, then their expected difference with $y \geq x$ is $E[y-x|y \geq x] = \frac{n-1}{3} = \Theta(n)$. The largest difference would be $y-x = n-1 = \Theta(n)$*

Proof.

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

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

Proposition 16. *The expected difference between c and a is linear in n independently of b, i.e. $E\left[c - a\right] = \frac{(n+1)}{2} = \Theta(n)$.*

Proof. The expected value of a is $E\left[a \mid b\right] = b/2 - (n/2 + 1)/2$, whereas $E\left[c \mid b\right] = b + (n/2 - b)/2$. It now follows that $E\left[c - a \mid b\right] = E\left[c \mid b\right] - E\left[a \mid b\right] = (n+1)/2$, independently of b.

□

Proposition 17. *Given x and y uniformly and independently distributed in R, then the probability that $x > y$ is $(1/2) - (1/2n)$. The probability that $x \leq y$ is $(1/2) + (1/2n)$.*

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

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

The probability of $x \leq y$ is hence $1 - (\frac{1}{2} - \frac{1}{2n}) = \frac{1}{2} + \frac{1}{2n}$ □

Lemma 1. *For search algorithms that use the fitness function only for direct comparisons of candidate solutions, the expected time for covering a branch ID_w is not higher than the expected time to cover any of its nested branches ID_z.*

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

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

□

6.3 Analysis of RS

Lemma 2. *Given g global optima in a search space of $|S|$ elements, then the expected time for RS to find an optimal solution is $E[T_{RS}] = |S|/g$.*

Proof. The probability of sampling an optimal solution is $p = g/|S|$. 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 $\Pr[T_{RS} = t] = (1-p)^{t-1} \cdot (p)$. Hence, the expected time for RS to find a global optimum is $E[T_{RS}] = (1/p) = |S|/g$. □

Theorem 1. *The expected time for RS to find an optimal solution to objective function f_8 is $2n^2 = \Theta(n^2)$ and for objective function f_{10} it is $\Theta(n)$. For all the other branches, the expected time is $\Theta(1)$.*

Proof. The search space is composed of n^3 elements. The proof simply follows by Lemma 2 and all the Propositions in Section 6.1. □

Theorem 2. *The probability that RS has found an optimal solution to objective function f_8 within n^3 iterations is exponentially large $1 - e^{-\Omega(n)}$.*

Proof. Using the inequality $(1 - 1/x)^x \leq e^{-1}$ and Theorem 1, we can see that $\Pr\left[T_{RS} > n^3\right] = \left(\left(1 - \frac{1}{2n^2}\right)^{2n^2}\right)^{n/2} \leq e^{-n/2}$. □

Proposition 18. *The expected time for RS to find an optimal solution to objective function f_8 when the branch distance is not used (i.e., Equation 1) is $2n^2 = \Theta(n^2)$.*

Proof. RS does not exploit any gradient in the fitness function. Therefore, the use of the branch distance does not make any difference. Proof hence follows from Theorem 1. □

6.4 Analysis of HC

Theorem 3. *The expected time for HC with neighbourhood N_d to find an optimal solution to objective function f_8 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 $a + b \leq c$ holds for all of them, we have $f_8(L) = \zeta + \omega(-t + \gamma)$. 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 13 and 14), that is $f_8(L') = \zeta + \omega(-t + 1 + \gamma)$. Because $f_8(L') > f_8(L)$, the points L are local optima.

Considering Propositions 13 and 14, 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 whether $a + b \leq c$ 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 algorithms 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 have already proved that all the points on 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 by Proposition 7.

A step is called successful if the new search point I' is accepted. The number of successful steps η 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 ψ 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 ψ, 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 ψ, the number η 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 η, the expected number of iterations for reaching an optimum is $\Theta(c - a)$. For Proposition 16, starting from a random point the expected number of iterations for reaching either a local or a global optimum is hence $\Theta(n)$.

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

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 $\Theta(n)$. □

Theorem 4. *The probability that HC with neighbourhood N_d has found an optimal solution to objective function f_8 within $k \cdot n^2$ iterations is exponentially large $1 - e^{-\Omega(n)}$, where k is a constant.*

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

Theorem 5. *The expected time for HC with neighbourhood N_d to find an optimal solution to objective function f_8 when the branch distance is not used (i.e., Equation 1) is $\Theta(n^2)$.*

Proof. Objective function in Equation 1 can assume only 3 values. Hence, before doing a restart, there can be at most 2 successful steps. Because the neighbourhood size is constant, then there can be at most a constant number of steps before doing a restart.

On the one hand, in the optimal scenario, all the solutions that are 2 steps away from a global optimum would have a gradient toward it. The probability of starting from one of these points would be $\frac{n}{2} \cdot 6^2 \cdot \frac{1}{n^3} = \frac{18}{n^2}$, hence the runtime would be $\Omega(n^2)$. On the other hand, in the worst scenario none of these points have a gradient, and HC would degenerate in a RS with runtime $O(n^2)$ (Theorem 1). Because the lower bound is equal to the upper bound, hence HC has runtime $\Theta(n^2)$.

□

Theorem 6. *The expected time for HC with neighbourhood N_d to find an optimal solution to objective function f_0 is $\Theta(n)$.*

Proof. Following by Proposition 17, we have $\frac{1}{4} \leq \Pr[x > y] < \frac{1}{2}$ for any $n > 1$. Hence, with constantly bounded probability, HC finds a solution in the first step, i.e. $\Theta(1)$ steps.

In the case in which $x \leq y$, there is a gradient to either increase x or to decrease y. By Proposition 15, the distance is $\Theta(n)$, hence $\Theta(n)$ steps are required.

Considering the constant bounds on the probability of the 2 different runtimes, the overall runtime is $\Theta(n)$.

□

6.5 Analysis of AVM

Lemma 3. *For any branch, if the probability that the random starting point is a global optimum is lower bounded by a positive constant $p > 0$, then AVM needs at most a constant number $\Theta(1)$ of restarts to find a global optimum.*

Proof. If we consider only the starting point, the AVM behaves as a random search, in which the probability of finding a global optimum is bigger than p. That can be described as a Bernoulli process (see Lemma 2), with expected number of restarts that is lower or equal than $1/p$. □

Theorem 7. *The expected time for AVM to find an optimal solution to the coverage of branches ID_0 and ID_1 is $O(\log n)$.*

Proof. Considering that the search is done for values of n bigger or equal than 8, then both the searches for ID_0 and ID_1 start with a random point that is a global optimum with a probability lower bounded by a positive constant (Proposition 17). Therefore, AVM needs at most a constant number of restarts (Lemma 3), independently of the presence and number of local optima.

For both branches, either the starting point is a global optimum, or the search will be influenced by the distance $x - y$ that is $\Theta(n)$ (Proposition 15). We hence analyse this latter case.

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

$\square$

Theorem 8. *The expected time for AVM to find an optimal solution to the coverage of branches ID_2, ID_3, ID_4 and ID_5 is $O(\log n)$.*

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

The type of predicates is the same of branches ID_0 and ID_1 (i.e., $>$), and the condition of the comparison is the same (i.e., $>$ on two input variables). The three input variables are uniformly and independently distributed in R, and by Proposition 15 the maximum distance among them is $\Theta(n)$. There are the same conditions of Theorem 7 apart from the fact that the variables could be swapped during the search, i.e. the fact that lines 5 and 8 are executed or not can vary during the search.

For branches ID_2 and ID_3, starting from z no variation of the executed code is done until the branch is covered. For branch ID_3, for either x or y a search starting from the maximum of them would result in no improvement of the fitness function. The minimum of x and y has a gradient to decrease, and while it does so the relation of their order is not changed. Hence, no variation of the executed code is done. On the other hand, for branch ID_2, the minimum has gradient to increase, but the pattern search would stop once it becomes the maximum of the two (e.g., $x > y$ if the search started on x with $x < y$). That happens in at most $O(\log n)$ steps because their difference is at most $\Theta(n)$ (Proposition 15). If the next variable considered by AVM is not z, then the above behaviour will happen again. However, the next variable will be necessarily z, hence we have at most $O(\log n)$ steps done 3 times, that still results in $O(\log n)$ steps.

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

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

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

□

Theorem 9. *The expected time for AVM to find an optimal solution to the coverage of branch ID_6 is $O(\log n)$.*

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

There is gradient to decrease a and b, and there is gradient to increase c. If the search starts from either a or b, in $O(\log n)$ steps of a pattern search the target variable assumes a negative value (the highest possible starting value is $n/2$). In particular, if the search starts from b, at a certain point the input variable representing b will instead represent a. Otherwise, it sufficient to increase c up to the value $a + b \leq n/2 + n/2 = n$, that can be done in $O(\log n)$ steps of a pattern search.

□

Lemma 4. *For objective f_8, given a starting point (a, b, c), before doing a restart AVM converges to an optimum $T = (t, t, t)$, where $a \leq t \leq c$.*

Proof. The variable representing a can only increase (Proposition 13), and it has a gradient to increase up to b, i.e., each succession of increments of a has better fitness till $a' = b$. Similarly, c cannot increase (Proposition 14), and it has a gradient to decrease down to b, and although b can change, b will still be in the interval $[a, c]$.

In the case of a pattern search that leads to a value k outside $[a, c]$, then that value has a gradient toward a if $k < a$ and toward c if $k > c$ (that can be easily proved with an induction on Propositions 13 and 14).

AVM modifies the same variable until it finds better solutions. Hence, before any other variables is modified, the current variable will have a value in the interval $[a, c]$.

Because, AVM accepts only strictly better solutions and it is deterministic, it will hence converge to a point $T = (t, t, t)$ in a finite number of iterations. □

Lemma 5. *For objective f_8, starting an exploratory search on any variable $g \in \{x, y, z\}$, after $\Theta(\log(n))$ steps of AVM the distance of g from the closest other variable is reduced by at least $2/3$.*

Proof. The variable representing a can only increase (Proposition 13) toward b, and the variable representing c can only decrease down to b (Proposition 14).

Considering the fitness function f_8, the variable representing b can increase toward c if $a + b \leq c$. Otherwise, it can be modified only in 2 cases:

$$b = a + 1\ ,$$
$$b = c - 1\ .$$

In these two cases, a single exploratory search makes either $a = b$ or $b = c$ in at most two steps.

If g represents either a or b, then a pattern search will increase its value, otherwise the value will be decreased (case c).

Let h be the closest variable to g. After a pattern search, we can have three cases: either $g = h$, or $g < h$ or $g > h$. Given (a_0, b_0, c_0) the ordered values of (x, y, z) when AVM starts to do a new exploratory search, Figures 5, 6 and 7 show examples of these three cases for g representing b. For Propositions 13 and 14, a pattern search ends when g assumes value bigger than c or lower than a.

To simplify the proof, assume that $g < h$ (the other case $g > h$ can be analysed in the same way by inverting the signs of the arithmetic operations and the inequalities).

Let g' be the value of g at the end of the pattern search. For Proposition 16, it follows that $|g - h| = \Theta(n)$. Hence, given s the number of steps for which AVM gets g as close as possible to h with $g_s < h$, then $s = \Theta(\log(n))$.

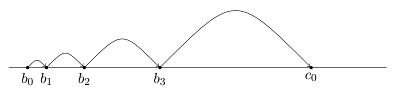

Figure 5: Example where $c_0 - b_0 = 15$, in which case $b_s = c_0$.

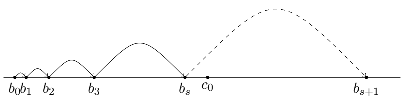

Figure 6: Example where $c_0 - b_0 = 17$, in which case b_{s+1} is not accepted, as indicated by the dashed arrow.

After s steps, the increment to the variable g is $\sum_{i=0}^{s} 2^i = 2^{s+1} - 1$. Let $k = 2^{s+1}$, hence after s steps the value of g is $g_s = g + k - 1$. Hence:

$$\begin{cases} g + k - 1 < h \, , \\ g + 2k - 1 > h \, . \end{cases}$$

There can be two cases: either $g' = g_s$, and that happens if g_{s+1} leads to a worse fitness, or otherwise $g' = g_{s+1}$. On the one hand, for $g' = g_s$, then it should be that:

$$h - g_s \leq g_{s+1} - h \, ,$$

which is the case for $k \geq \frac{2}{3}(h - g + 1)$. On the other hand, for $g' = g_{s+1}$ it should be:

$$h - g_s > g_{s+1} - h \, ,$$

and the highest value that g' can have is $g_{s+1} = g + \frac{4}{3}(h - g + 1)$. Therefore it follows that $|h - g'| \leq \frac{1}{3}|h - g|$. □

Lemma 6. *The lower and upper bounds of the expected time of AVM for converging to an optimum for objective f_8 is $\Omega(\log n)$ and $O((\log n)^2)$.*

Proof. By Lemma 4, AVM converges to a solution $T = (t, t, t)$ before doing a restart if T is not a global optimum.

Starting the search on any variable $g \in \{x, y, z\}$, after $\Theta(\log(n))$ steps, its distance from the closest other variable is reduced by at least $2/3$ (Lemma 5). AVM does modifications on the same variable till improvements can be found. Figure 8 shows an example of a new pattern search on the same variable.

By applying Lemma 5 recursively, because the distance is reduced at least by $2/3$ at each pattern search, then after $O(\log n)$ pattern searches we have g that is equal to another variable, which we

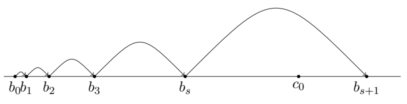

Figure 7: Example where $c_0 - b_0 = 25$, in which case b_{s+1} is accepted.

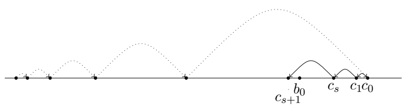

Figure 8: Example in which $c_0 - b_0 = 6$. In that case c_{s+1} is accepted. To note that is the continuation of the search done in figure 7, which is represented in dotted arrows. The former b_{s+1} has become the new c_0, whereas the old c_0 is now the new b_0.

call h. Therefore, that in expectation would happen after a number of steps with bounds $\Omega(\log n)$ and $O((\log n)^2)$.

However, the distance $|g - h|$ gets reduced at each search. Hence, the upper bound for the steps is:

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

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

Once we get $g = h$, any modification of g and h would not lead to any better value for fitness function f_8. The modification of the third variable will follow the same behaviour of g, and after another i expected iterations with same bounds, AVM converges to T. Because only a constant number of variables is modified (i.e., 2), the lower and upper bounds of the expected time of AVM for converging to T is $\Omega(\log n)$ and $O((\log n)^2)$.

□

Lemma 7. *For objective f_8, in expectation, AVM needs a constant number $\Theta(1)$ of restarts to reach a global optimum.*

Proof. If $a > 0$, then AVM converges to an optimum (Lemma 4) which is global (Proposition 7). The probability of this event is $\frac{1}{8}$. Considering restarts as a geometric process, the expected number of restarts is less or equal to 8 (Lemma 3). □

Theorem 10. *The lower and upper bounds of the expected time of AVM to find an optimal solution to objective f_8 is $\Omega(\log n)$ and $O((\log n)^2)$.*

Proof. By Lemma 7 there are $\Theta(1)$ restarts, and by Lemma 6, each search requires in expectation $\Omega(\log n)$ and $O((\log n)^2)$ steps regardless it takes to a global or local optimum. □

Theorem 11. *The probability that AVM has found an optimal solution to objective function f_8 within $k \cdot n \cdot (\log n)^2$ iterations is exponentially large $1 - e^{-\Omega(n)}$, where k 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 Lemma 6, AVM has reached a local optimum within $k \cdot (\log n)^2$ iterations, for some constant k. 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 $(1-p)^n = e^{-\Omega(n)}$. □

Theorem 12. *The expected time for AVM to find an optimal solution to objective function f_8 when the branch distance is not used (i.e., Equation 1) is $\Theta(n^2)$.*

Proof. The proof follows the same discussion as in the proof of Theorem 5 for HC. The difference is that the visited neighbourhood might be larger due to a possible pattern search. However, the number of visited solutions is still bounded by a constant, so the runtime is like the one of HC, i.e. $\Theta(n^2)$. □

Theorem 13. *The expected time for AVM to find an optimal solution to the coverage of branch ID_7 is $O((\log n)^2)$.*

Proof. The branch ID_8 is nested to branch ID_7, hence by Lemma 1 and Theorem 10 the expected time is $O((\log n)^2)$. □

Theorem 14. *The expected time for AVM to find an optimal solution to the coverage of branch ID_9 is $O((\log n)^2)$.*

Proof. By Theorem 13, the branch ID_7 can be covered in $O((\log n)^2)$ steps. The branch ID_9 (that is nested to ID_7), will be covered if $\neg(a = b \wedge b = c)$. If that predicate is not true, a single exploratory search of AVM makes it true because it is just sufficient to either increase or decease any input variable by 1. The only case in which this is not possible is for $I = (1, 1, 1)$, because it is the only solution that satisfies $a = b \wedge b = c \wedge a - 1 + b \leq c \wedge a + b \leq c + 1 \wedge a + b > c$. In that case, a restart is done.

With a probability that is lower bounded by a constant, in a random starting point each input variable is higher than $n/4$. In that case, $a + b > c$, because $(n/4) + 1 + (n/4) + 1 > (n/2)$. By Lemma 3, we need only $\Theta(1)$ restarts.

□

Theorem 15. *The expected time for AVM to find an optimal solution to the coverage of branch ID_{10} is $O((\log n)^2)$.*

Proof. By Theorem 14, the branch ID_9 can be covered in $O((\log n)^2)$ steps. The branch ID_{10} (that is nested to ID_9), will be covered if only two input variables are equal (and not all three equal to each other at the same time).

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

The other case in which all the three variables are different is quite complex to analyse. Instead of analysing it directly, we prove the runtime by a comparison with the behaviour of AVM on the branch ID_8 (proved in Theorem 10).

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

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

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

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

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

By Theorem 10, the expected time for covering branch ID_8 is $O((\log n)^2)$. Because we proved that the coverage of ID_8 takes more time than the coverage of ID_{10}, then the expected runtime for covering ID_{10} is $O((\log n)^2)$.

□

Theorem 16. *The expected time for AVM to find an optimal solution to the coverage of branch ID_{11} is $O((\log n)^2)$.*

Proof. By Theorem 14, the branch ID_9 can be covered in $O((\log n)^2)$ steps. The branch ID_{11} (that is nested to ID_9), will be covered if $a \neq b \wedge a \neq c \wedge b \neq c$. In the moment that the branch ID_9 is executed, then the three variables cannot assume all the same value (otherwise the branch ID_8 would have been executed). If that predicate is not true, a single exploratory search of AVM makes it true because it is just sufficient to increase by 1 any of the two variables that have same values.

□

6.6 Analysis of (1+1) EA

In contrast to the other algorithms we have analysed so far, the (1+1) EA uses binary strings to represent solutions. We denote the ith bit of bitstring x by x_i, the length of a bitstring x by $\ell(x)$, and the concatenation of two bitstrings x and y either by $x \cdot y$ or xy. Test cases for the triangle classification program will be encoded as bitstrings $I \in \{0,1\}^{3m}$, which for notational convenience will be denoted as a triple $\{x, y, z\}$ where $x := I_1 \cdots I_m$, $y := I_{m+1} \cdots I_{2m}$ and $z := I_{2m+1} \cdots I_{3m}$. Here, we will consider unsigned integers, where a bitstring x of length $\ell(x)$ has integer value $\mathrm{bin}(x) := \sum_{i=1}^{\ell(x)} x_i \cdot 2^{m-i}$.

The (1+1) EA is a comparison-based algorithm in the sense that the decision whether to replace the current search point x with a new search point x' only depends on the ordering of x and x' with respect to the fitness function f, and not on the actual fitness value of search point x'. Hence, the (1+1) EA will not change behaviour if the fitness function f is replaced by another fitness function g where for all bitstrings x and y, $f(x) < f(y)$ if and only if $g(x) < g(y)$. To simplify the notation, we will therefore use the function $f(x, y, z) = |x - y| + |y - z| = \max\{x, y, z\} - \min\{x, y, z\}$ instead of function f_8. Furthermore, instead of directly analysing function f_0, we consider the function $f(x, y) = y - x$.

To analyse the progress of the (1+1) EA towards the optimum of function f_8, we define the block length of a search point, and use this is a potential function.

Definition 9 (Block length). *Let x, y and z be three bitstrings of length m with longest common prefix σ. The* prefix-length *of the triple x, y, z is the length $\ell(\sigma)$ of the prefix σ, and the* block length *is the largest integer s such that $x, y, z \in \{\sigma 1 0^s \alpha, \sigma 0 1^s \beta \mid \alpha, \beta \in \{0,1\}^{m-\ell(\sigma)-1-s}\}$.*

Among bitstrings with the same prefix length, the value of Korel's distance function decreases almost monotonically with the block length.

Lemma 8. *On objective function f_8, if bitstrings x, y and z have length m, longest common prefix σ, and block length s, then $2^{r-s} + 1 \leq f(x, y, z) \leq 2^{r-s+2} - 1$, where $r = m - \ell(\sigma) - 2$.*

Proof. The initial part of the triangle classification program assigns the minimal value of x, y, z to variable a, and the maximal value of x, y, z to c. Hence, we have $f(x, y, z) = \text{bin}(c) - \text{bin}(a)$. If the bitstring representations of a and c can be written on the form $a = \sigma 01^s x\beta$ and $b = \sigma 10^s 1\alpha$, then $\text{bin}(\sigma 10^s 1\alpha) - \text{bin}(\sigma 01^s x\beta) = 2^{r-s} \cdot (3 - x) + \text{bin}(\alpha) - \text{bin}(\beta)$. Otherwise, it must be possible to write the bitstrings on the forms $c = \sigma 10^s 1\alpha$ and $a = \sigma 01^s x\beta$, in which case $\text{bin}(\sigma 10^s x\alpha) - \text{bin}(\sigma 01^s 0\beta) = 2^{r-s} \cdot (2 + x) + \text{bin}(\alpha) - \text{bin}(\beta)$. Furthermore, the difference $\text{bin}(\alpha) - \text{bin}(\beta)$ is at least $-2^{r-s} + 1$ and at most $2^{r-s} - 1$. So in all cases, $2^{r-s} + 1 \leq \text{bin}(c) - \text{bin}(a) \leq 2^{r-s+2} - 1$. □

The probability of increasing the prefix-length is small.

Lemma 9. *Consider (1+1) EA on objective function f_8 with bitstring length $3m$. If the prefix-length of the current search point is s, then the probability that the current search point in the following iteration has prefix length $s + i$ for any $i > 0$, is less than m^{-l-1}, where $l := \min\{i, s\}$.*

Proof. The case where $i > s$ is trivial, because one of the bitstrings differs in $s + 1$ positions, and it is therefore necessary to flip at least these $s + 1$ bits to increase the length of the common prefix bits by i.

Consider the case where $i \leq s$. Without loss of generality, assume that the current search point is of the form $\{\sigma 10^{i-1}00^{s-i}\alpha, \sigma 01^{i-1}11^{s-i}\beta, \sigma 01^{i-1}11^{s-i}\kappa\}$, where $\ell(\alpha) = \ell(\beta) = \ell(\kappa) = m - \ell(\sigma) - s - 1$. By Lemma 8, the fitness value of this search point is no more than $2^{\ell(\alpha)+1} - 1$. To increase the prefix-length by i, it is necessary to flip at least i bits after σ in at least one of the bitstrings. Assume that the mutated search point is accepted without also flipping the next bit in position $\ell(\sigma) + i + 1$. Then the search point will be of the form $\{\sigma 01^{i-1}0\alpha', \sigma 01^{i-1}1\beta', \sigma 01^{i-1}1\kappa'\}$, where $\ell(\alpha') = \ell(\beta') = \ell(\kappa') = m - \ell(\sigma) - i - 1 \geq \ell(\alpha)$. The fitness of this search point is at least $\text{bin}(\sigma 01^{i-1}1\beta') - \text{bin}(\sigma 01^{i-1}0\alpha') \geq 2 \cdot 2^{\ell(\alpha')} - \text{bin}(\alpha') \geq 2^{\ell(\alpha')} + 1$, which is strictly larger than the original search point and therefore contradicts that the search point was accepted by (1+1) EA. □

We are now in position to lower bound the runtime of (1+1) EA on the equilateral branch of the triangle inequality program. We only count the runs where the algorithm reaches the search point $\{10^{m-1}, 01^{m-1}, 01^{m-1}\}$ before the optimum has been found, and optimistically assume that all other runs are finished in 0 iterations.

Lemma 10. *With constant probability $p > 0$, (1+1) EA will reach a search point on the form $\{10^{m-1}, 01^{m-1}, 01^{m-1}\}$ before reaching the global optimum of objective function f_8.*

Proof. With probability $3/2^9$, the initial search point is on the form $x, y, z = \{100\alpha, 011\beta, 011\kappa\}$. Clearly, this search point does not satisfy the predicate `(a + b <= c)`, hence the remaining of the search consists in trying to reach the equilateral branch. Let the block length of this search point be $s \geq 2$. The probability that the prefix length increases, is by Lemma 9 no more than $1/m^2$. In the following, we will analyse a duration of km^2 iterations, where k will be specified later, and assume that the prefix length will never increase during this period, an event which happens with probability at least $(1 - 1/m^2)^{km^2} = \Omega(1)$.

By Lemma 8, if the block length increases to $s + 2$, then all future search points will have block length at least $s + 1$. We call a *trial* a sequence of search points where the current block length s is increased or decreased. The trial lasts until either the block length has been reduced to $s - 1$, or the block length has been increased from s to $s + 2$, in which case the trial is called *successful*. We now estimate the probability of a successful trial.

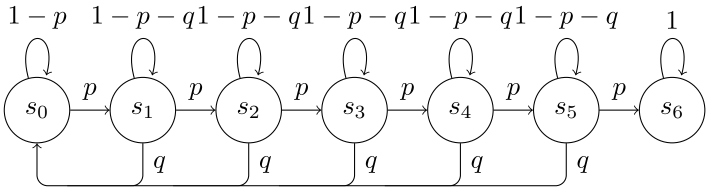

Figure 9: Markov chain in the proof of Lemma 10.

In order to decrease the block length to $s-1$, it is necessary to flip at least one of the bits in position $1+s$ of x, y or z, an event which happens with probability no more than $1/m$. In order to increase the block length to $s+2$, it suffices to flip the two right-most 1-bits in α, the two right-most 0-bits in β and the two right-most 0-bits in κ. Each of these 6 bit-flips happen with a probability of $1/3m$ in any iteration. When all 6 bits have been flipped, the block length must necessarily have been increased to at least $s+2$. To analyse the stochastic process behind these bitflips, we construct a Markov chain corresponding to the number of those bits that have the "correct" value. We pessimistically assume that at most one bit can be flipped correctly in each iteration, and if the block length is reduced, or one of these bits are flipped back, then all bits are lost. The Markov chain is depicted in Figure 9, where state s_i corresponds to i correct bits, and the values of the variables occurring in the state transition probabilities are defined as $p := 1/3em$ and $q := 1/m$. With some algebraic manipulation, it is easy to see that the expected hitting time from state s_0 to state s_6 is $7/p + 6q/p^2 = km$ for some constant k. The block length must be increased at most m times, hence the expected time until the search point is on the form $\{10^{m-1}, 01^{m-1}, 01^{m-1}\}$ is no more than km^2, and by Markov's inequality, the probability that this search point is reached within $2km^2$ steps is at least $1/2$.

The probability of increasing the block length within km^2 iterations is by union bound no more than k. Hence, the probability that the search point $\{10^{m-1}, 01^{m-1}, 01^{m-1}\}$ has been obtained before the prefix length is increased is at least $3/2^9 \cdot (1/2)/(1/2+k) > 0$. □

Theorem 17. *The expected running time of (1+1) EA on objective function f_8 with integers in the interval $[0, n)$ represented in binary is $\Omega((\log_2 n)^5)$.*

Proof. Define $m := \log_2 n$. We define a typical run as a run where the EA reaches the search point $\{10^{m-1}, 01^{m-1}, 01^{m-1}\}$, which has function value 1, before reaching the global optimum. By Lemma 10, there is a constant probability that the run is typical. We will lower bound the expected runtime by the expected number of iterations needed to find the global optimum starting from this position. By Lemma 8, the only non-optimal search points that will be accepted from this point are on one of the forms $\{\sigma 10^s, \sigma 01^s, \sigma 01^s\}$ and $\{\sigma 01^s, \sigma 10^s, \sigma 10^s\}$. All other search points have block lengths at most $m - \ell(\sigma) - 2$, and therefore function values at least 2.

In order to reach the global optimum, it is necessary to increase the prefix length to m. We use drift analysis [23] to bound expected runtime until this happens, using m minus the prefix length as distance function. In order to decrease the distance by $i > 0$, it necessary to flip at least $i+3$ bits simultaneously. Hence, the expectation of the drift Δ in each iteration is at most $\sum_{i=1}^{m} i \cdot m^{-i-3} = O(m^{-4})$. The initial distance from the optimum is $B = \Theta(m)$, hence the expected runtime is at least $B/E[\Delta] = \Omega(m^5)$. □

Theorem 18. *The expected running time of (1+1) EA on objective function f_8 with integers in the interval $[0, n)$ represented in binary is $O((\log_2 n)^5)$.*

Proof. Define $m := \log_2 n$. We divide a run of the EA into three phases. The first phase begins with the initial iteration and lasts until $a + b > c$ holds. The second phase ends when the search point has block length $m - \ell(\sigma) - 1$. The third phase lasts until a search point $a = b = c$, i.e. the global optimum, has been found.

For the first phase to end, it suffices to wait for a search point on the form $a = 1\alpha, b = 1\beta, c = 1\kappa$, because $\text{bin}(1\alpha) - \text{bin}(1\beta) - \text{bin}(1\kappa) \leq -1$ for any α, β and κ. Such search points are obtained by flipping at most 3 bits simultaneously, hence the expected duration of the first phase is bounded by $O(m^3)$.

The runtime of the second phase can be analysed using drift analysis, similarly to the proof of Lemma 10. Hence, the duration of the second phase is bounded from above by $O(m^2)$.

For the third phase, we apply drift analysis as in the proof of Theorem 17. For reasons that will be explained later, we start analysing the algorithm after iteration m^3. In the worst case, the search point has prefix-length 0 at this point in time. In a given iteration, we distinguish between two complementary events. Let $\mathcal{E}$ be the event that the search point is on one of the two forms $\{\sigma 1 \cdot 10^s, \sigma 1 \cdot 01^s, \sigma 1 \cdot 01^s\}$ or $\{\sigma 0 \cdot 10^s, \sigma 0 \cdot 10^s, \sigma 0 \cdot 01^s\}$, and let the complementary event $\overline{\mathcal{E}}$ denote the event that the search point is on one of the two forms $\{\sigma 0 \cdot 10^s, \sigma 0 \cdot 01^s, \sigma 0 \cdot 01^s\}$ or $\{\sigma 1 \cdot 10^s, \sigma 1 \cdot 10^s, \sigma 1 \cdot 01^s\}$.

In the case of event $\mathcal{E}$, it is necessary to flip 5 bits to reduce the prefix-length by 1. Hence, the conditional expected drift becomes $E\left[\Delta \mid \mathcal{E}\right] = \Omega(m^{-4}) - O(m^{-5}) = \Omega(m^{-4})$. In the complementary event $\overline{\mathcal{E}}$, it is necessary to flip 4 bits to reduce the prefix length by 1, but the probability of increasing the prefix-length is no larger than the probability of increasing the prefix-length. Hence, we have $E\left[\Delta \mid \overline{\mathcal{E}}\right] \geq 0$. We first claim that $\Pr\left[\mathcal{E}\right] = \Omega(1)$. If this claim holds, we have unconditional expected drift $E\left[\Delta\right] = \Omega(m^{-4})$, and the expected duration of the second phase is $O(m^5)$.

We finally show that the claim $\Pr\left[\mathcal{E}\right] = \Omega(1)$ holds. Let the current iteration number be t. Let random variable $X \in \{0, 1\}$ denote the value of the last bit of the common prefix σ in iteration t. We will analyse the behaviour of variable X in the time interval $t - m^3$ to t conditional on the event $\mathcal{F}$ that the prefix length is constant in this period. The probability of increasing the prefix-length in any iteration is less than m^{-4}, so the probability of event $\mathcal{F}$ is at least $(1 - m^{-4})^{m^3} = \Omega(1)$. Given that the prefix length is constant, the probability of changing the value of X in any iteration is $\Theta(m^{-3})$. If $X_0 = 0$, then the probability that $X_{m^3} = 1$ is at least $m^3 \cdot (1 - m^{-3})^{m^3 - 1} m^{-3} = \Omega(1)$. If $X_0 = 1$, then the probability that $X_{m^3} = 1$ is at least $m^3 \cdot (1 - m^{-3})^{m^3} = \Omega(1)$. Hence, the probability of event $\mathcal{E}$ is $\Omega(1)$.

The expected duration of all three phases is therefore $O(m^5)$. $\square$

We now turn to the runtime of (1+1) EA on the first branch in the program ID_0, in which case Korel's distance function gives the minimisation objective $y - x$.

Theorem 19. *The expected runtime of (1+1) EA using either branch distance and approach level (i.e. objective function f_0), or only approach level with integers in the interval $[0, n)$ on the covering of branch ID_0 is $\Theta(\log_2 n)$.*

Proof. Define $m := \log_2 n$. We will focus on the leading bits of x and y only. For the upper bound, we pessimistically assume that the predicate cannot be satisfied when $x_1 = 0$ and $y_1 = 0$, however it is clear that the predicate is necessarily satisfied when $y_1 = 0$ and $x_1 = 1$. In the worst case, we start with $y_1 = 1$ and $x_1 = 0$. From this state, it suffices to wait for a sequence of two bit-flips which do not flip the same variable twice. The expected time until one bit-flip occurs is less than em, and the expected time until the right bit-flip sequence occurs is no more than $8em$.

For the lower bound, we note that there is a constant probability that the initial search point has $y_1 = 1$ and $x_1 = 0$, and that the probability that none of these are flipped within m iterations is at least $(1 - 2/m)^m = \Omega(1)$. Hence, the runtime is lower bounded by $\Omega(m)$. $\square$

Note that the previous theorems for (1+1) EA do not consider negative values for the three input variables. In order to consider negative values, it is necessary to extend the runtime analysis of (1+1) EA.

However we conjecture that if signed integers are represented using a separate sign bit, the asymptotic runtime results will be the same. In the following discussion, we will assume that this conjecture holds.

7 Discussion

Table 4 summarises the empirical results and the theoretical ones for branches ID_8 and ID_0.

For branch ID_8, the theoretical analyses show that RS has the worst runtime, whereas AVM has the best. Not only can AVM find a solution in an efficient way, but we also proved that it has an extremely low probability of not finding an optimal solution in a reasonable time (Theorem 11). In other words, it is very unlikely that AVM will run for a long time (depending on n) without finding an optimal solution. This is an important result which cannot be obtained with empirical studies. In fact, given any number k of experiments, we can say only little about the worst case scenario, and how often it happens. For example, the experiments might show a low runtime, although actually it can happen that with a small probability the runtime is enormous (and hence in average the runtime is enormous), but k was not large enough to show it.

We proved that the runtime of AVM is $O(\log(n)^2)$ on all the branches of TC. This is necessary and sufficient to state that AVM has the best runtime among the analysed search algorithms because all the other heuristics have a strictly worse runtime on at least one branch of TC.

It is not a surprise that AVM has the best runtime. The fitness landscape is relatively simple even for the most difficult branch ID_8. The big jumps of the pattern search quickly bring AVM to a global optimum. AVM avoids the local optima that exist in this problem through restarts. This is particularly helpful when the local optima are far from the global optima, as is the case on this problem. Although the search landscape can be considered easy, analysing the actual behaviour of search algorithms on this landscape is far from been trivial.

The empirical results in Table 3 show that, even with low values of n, most of the time the correct runtimes for RS, HC and AVM are correctly inferred by the regression analysis. On the other hand, the empirically estimated runtime for (1+1) EA in Table 3 is incorrect because the correct model was not used for the regression.

We proved that (1+1) EA on target ID_8 has a runtime of $\Theta((\log n)^5)$. That was a surprise for us, because we were expecting something similar to the runtime of either AVM or HC. Although the empirical results in Table 3 clearly show that (1+1) EA is much slower than HC for low values of n, the asymptotic runtime of (1+1) EA is strictly better. In other words, (1+1) EA is faster than HC for large values of n. If we do not consider the constants, it will happen for values n for which $n > (\log n)^5$ is true, i.e. for $n \geq 5 \cdot 10^6$. In our experiments we tested till $n = 8192$, and the performances of (1+1) EA for that value (and below) are poor because $(\log 8192)^5 = 371293$, which is much higher than 8192. This is a clear example of an algorithm that empirically seems to perform relatively poorly compared with another algorithm, but actually has better scalability.

Because the first regression analysis on target ID_8 did not include the correct runtime model for (1+1) EA, supplemental regression analysis was carried out with the following large set of models $\rho n^t \log(n)^v$, where $t \in \{0, 1, 2\}$ and $v \in \{0, \ldots, 10\}$. The results are shown in Table 5. Unfortunately, the correct model could still not be inferred from the empirical results. More runs and empirical data is needed.

SSGA was not analysed theoretically. It would be difficult to estimate a runtime for this algorithm based on the empirical results in Table 3. In fact, with size till 1024 the empirically estimated runtime is $\rho \log(n)^2$, but if we add 30 new points obtained with size 2048, then the inferred model is ρn. Adding another set of experiments (i.e., 30 points with size 4096) changes the estimated model to ρn^2, and then again to $\rho \log(n)^2$ when we consider size 8192. It is unclear what model would be inferred with further empirical data on even larger instance sizes. In general, we cannot answer this question with empirical

Table 4: Summary of the empirically (Emp.) and theoretically (Th.) obtained runtimes for target branches ID_8 and ID_0. BD stands for “Branch Distance”.

Target Branch	BD	RS		HC		AVM		(1+1) EA		(μ+1) SSGA	
		Emp.	Th.	Emp.	Th.	Emp.	Th.	Emp.	Th.	Emp.	Th.
ID_8	T	$\rho n^2 \log(n)$	$\Theta(n^2)$	ρn	$\Theta(n)$	$\rho \log(n)$	$O(\log(n)^2)$	ρn	$\Theta(\log(n)^5)$	$\rho \log(n)^2$	–
	F	ρn^2	$\Theta(n^2)$	$\rho n^2 \log(n)^2$	$\Theta(n^2)$	ρn^2	$\Theta(n^2)$	$\rho n^2 \log(n)$	–	ρn^2	–
ID_0	T	ρ	$\Theta(1)$	$\rho \log(n)^2$	$\Theta(n)$	$\rho \log(n)$	$\Theta(\log(n))$	$\rho \log(n)$	$\Theta(\log(n))$	ρ	–

Table 5: Result of experiments for branch target ID_8 with a larger set of models. Data were collected with different values of n.

Max n	$(1+1)$EA
16	$0.0120 \log(n)^7$
32	$0.0120 \log(n)^7$
64	$0.0000\, n^2 \log(n)^{10}$
128	$0.0023\, n \log(n)^7$
256	$0.0009 \log(n)^{10}$
512	$0.0063 \log(n)^9$
1024	$0.0839\, n \log(n)^5$
2048	$0.0716 \log(n)^8$
4096	$7.8240 \log(n)^6$
8192	$6279.0000\, n$

studies, because for each tested size there will be always a bigger one that we have not tested. Moreover, as was the case for (1+1) EA, the correct runtime model may not be among those that were tested.

By Theorem 1, RS is very efficient (i.e., $\Theta(1)$) for most of the branches. It is only for ID_8 that it gets the runtime $\Theta(n^2)$. For branch ID_{10} it has the runtime $\Theta(n)$, which is equal with the overall runtime of HC (assuming that HC has not worse runtime on the other 10 branches besides ID_0 and ID_8 that we have not theoretically analysed). This is very important to keep in mind. In fact, when we test software, not all of the testing tasks are necessarily difficult to carry out. Some of them can be easy. We need sophisticated techniques only for the difficult testing problems, because on simple problems they can give worse results. Because a priori it is very difficult to understand whether a problem is either easy or difficult, hybrid strategies are required. For example, doing random testing before applying more sophisticated techniques is likely a wise choice.

Branch distance has been designed to improve the performance of search algorithms. When we do not use the branch distance in the fitness function (e.g., as in Equation 1), the search practically degenerates to random search. Counterintuitively, as we explained, on easy instances that can even produce better results.

8 Conclusion

In this paper we have illustrated how runtime analysis can be applied in SBSE, and we have advocated its importance.

On one hand, it was shown that empirical results can be misleading. Theoretical analyses can give insights into the behaviour of the search algorithms that empirical studies cannot give. This insight can be very useful in the long term to design new algorithms that push the boundaries of the current state of art. The obtained theoretical results are stronger than empirical ones. They can be used in the future any time the analysed case study is employed in new empirical analyses where novel techniques are validated and compared.

On the other hand, theoretical analyses have their own limits. It can be hard to analyse the runtime, and it might be infeasible to show for large and complex software (e.g., it is required that all the optima are known in advance). Besides, proofs for a particular testing problem are difficult to generalise to another case study. Theoretical analyses are hence not meant to replace empirical studies.

It is important to note that theoretical analyses are not meant to be used on new problems for which we are looking for a solution. Their goal is to get insight knowledge and to compare the behaviour of search algorithms. This is similar to the type of empirical analysis in which different search algorithms are applied (and then compared) to known problems whose optimal solutions have been already found [62, 3].

In future work, we are planning to formally analyse more search algorithms that are commonly used in SBSE, like for example Simulated Annealing [54] and Genetic Algorithms [46]. This can be very challenging, but their theoretical analyses in traditional combinatorial problems are appearing in recent years [44]. Other problems in SBSE (e.g., requirement engineering [4]) should be addressed as well. Regarding software testing, general theoretical results that can be applied to entire classes of software are worth pursuing.

9 Acknowledgements

This work is supported by EPSRC grant EP/D052785/1.

References

[1] A. Arcuri. Full theoretical runtime analysis of alternating variable method on the triangle classification problem. In *To appear in International Symposium on Search Based Software Engineering (SSBSE)*, 2009.

[2] A. Arcuri, P. K. Lehre, and X. Yao. Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem. In *International Workshop on Search-Based Software Testing (SBST)*, pages 161–169, 2008.

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

[4] A. J. Bagnall, V. J. Rayward-Smith, and I. M. Whittley. The next release problem. *Information and Software Technology*, 43(14):883–890, 2001.

[5] B. Beizer. *Software Testing Techniques*. Van Nostrand Rheinhold, New York, 1990.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, New York, NY, 2nd edition, 2001.

[7] 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.

[8] K. A. Derderian, R. M. Hierons, M. Harman, and Q. Guo. Automated unique input output sequence generation for conformance testing of fsms. *The Computer Journal*, 49(3):331–344, 2006.

[9] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[10] B. Doerr, C. Klein, and T. Storch. Faster evolutionary algorithms by superior graph representation. In *Proceedings of the 1st IEEE Symposium on Foundations of Computational Intelligence (FOCI'2007)*, pages 245–250, 2007.

[11] S. Droste, T. Jansen, and I. Wegener. On the optimization of unimodal functions with the (1 + 1) evolutionary algorithm. In *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 13–22, London, UK, 1998. Springer-Verlag.

[12] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276:51–81, 2002.

[13] 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.

[14] T. Friedrich, N. Hebbinghaus, and F. Neumann. Rigorous analyses of simple diversity mechanisms. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO'07)*, pages 1219–1225.

[15] O. Giel. *Zur Analyse von randomisierten Suchheuristiken und Online-Heuristiken*. PhD thesis, Universität Dortmund, 2005.

[16] O. Giel and P. K. Lehre. On the effect of populations in evolutionary multi-objective optimization. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 651–658, New York, NY, USA, 2006. ACM.

[17] O. Giel and I. Wegener. Evolutionary algorithms and the maximum matching problem. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2003)*, pages 415–426, 2003.

[18] F. Gruenberger. Program testing: The historical perspective. *Program Test Methods*, pages 11–14, 1973.

[19] Q. Guo, R. M. Hierons, M. Harman, and K. A. Derderian. Computing unique input/output sequences using genetic algorithms. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES'2003)*, volume 2931 of *LNCS*, pages 164–177, 2004.

[20] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pages 342–357, 2007.

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

[22] 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.

[23] J. He and X. Yao. A study of drift analysis for estimating computation time of evolutionary algorithms. *Natural Computing*, 3(1):21–35, 2004.

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

[25] J. Jägersküpper. *Probabilistic Analysis of Evolution Strategies Using Isotropic Mutations*. PhD thesis, Universität Dortmund, 2006.

[26] T. Jansen. On the brittleness of evolutionary algorithms. In *Foundations of Genetic Algorithms*, volume 4436 of *Lecture Notes in Computer Science*, pages 54–69. Springer Berlin / Heidelberg, 2007.

[27] T. Jansen and I. Wegener. Real royal road functions–where crossover provably is essential. *Discrete Applied Mathematics*, 149(1-3):111–125, 2005.

[28] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.

[29] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[30] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from b formal models: Research articles. *Software Testing, Verification and Reliability*, 14(2):81–103, 2004.

[31] P. Lehre and X. Yao. Crossover can be constructive when computing unique input output sequences. In *Proceedings of the International Conference on Simulated Evolution and Learning (SEAL)*, pages 595–604, 2008.

[32] P. K. Lehre and X. Yao. Runtime analysis of (1+1) ea on computing unique input output sequences. In *Congress on Evolutionary Computation (CEC)*, pages 1882–1889, 2007.

[33] P. K. Lehre and X. Yao. Runtime analysis of (1+1) EA on computing unique input output sequences. In *Proceedings of 2007 IEEE Congress on Evolutionary Computation (CEC'07)*, pages 1882–1889, 2007.

[34] J. C. Lin and P. L. Yeh. Automatic test data generation for path testing using GAs. *Information Sciences*, 131(1-4):47–64, 2001.

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

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

[37] 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.

[38] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

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

[40] F. Neumann. *Combinatorial Optimization and the Analysis of Randomized Search Heuristics*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2006.

[41] F. Neumann and I. Wegener. Randomized local search, evolutionary algorithms, and the minimum spanning tree problem. *Theoretical Computer Science*, 378(1):32–40, 2007.

[42] F. Neumann and C. Witt. Runtime analysis of a simple ant colony optimization algorithm. In *In Proceedings of The 17th International Symposium on Algorithms and Computation (ISAAC 2006)*, number 4288 in LNCS, pages 618–627, 2006.

[43] P. Oliveto, J. He, and X. Yao. Analysis of population-based evolutionary algorithms for the vertex cover problem. In *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI'08), Hong Kong, June 1-6, 2008*, 2008.

[44] 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):281–293, 2007.

[45] P. S. Oliveto and C. Witt. Simplified drift analysis for proving lower bounds in evolutionary computation. In *Proceedings of Parallel Problem Solving from Nature (PPSN'X)*, number 5199 in LNCS, pages 82–91, 2008.

[46] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282, 1999.

[47] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[48] 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.

[49] G. Rudolph. Finite markov chain results in evolutionary computation: A tour d'horizon. *Fundamenta Informaticae*, 35(1):67–89, 1998.

[50] J. Scharnow, K. Tinnefeld, and I. Wegener. Fitness landscapes based on sorting and shortest paths problems. In *Proceedings of 7th Conf. on Parallel Problem Solving from Nature (PPSN–VII)*, number 2439 in LNCS, pages 54–63, 2002.

[51] T. Storch and I. Wegener. Real royal road functions for constant population size. *Theoretical Computer Science*, 320(1):123–134, 2004.

[52] D. Sudholt and C. Witt. Runtime analysis of binary pso. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 135–142, New York, NY, USA, 2008. ACM.

[53] W. Visser, C. S. Pasareanu, and R. Pelànek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.

[54] 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.

[55] I. Wegener. Methods for the analysis of evolutionary algorithms on pseudo-boolean functions. Technical Report CI-99/00, Universität Dortmund, 2000. Reihe computational intelligence collaborative research center 531.

[56] I. Wegener and C. Witt. On the analysis of a simple evolutionary algorithm on quadratic pseudo-boolean functions. *Journal of Discrete Algorithms*, 3(1):61–78, 2005.

[57] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[58] C. Witt. Worst-case and average-case approximations by simple randomized search heuristics. In *In Proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science (STACS'05)*, number 3404 in LNCS, pages 44–56, 2005.

[59] C. Witt. Population size versus runtime of a simple evolutionary algorithm. *Theoretical Computer Science*, 403(1):104–120, 2008.

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

[61] M. Woodward. Editorial: A test of time across the generations. *Software Testing, Verification & Reliability*, 14(2):79–80, 2004.

[62] 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.