UNIVERSITY OF BIRMINGHAM
University of Birmingham
Research at Birmingham

Runtime analysis of the (1+1) EA on computing unique input output sequences

Lehre, Per Kristian; Yao, Xin

DOI:
10.1016/j.ins.2010.01.031

License:
Creative Commons: Attribution (CC BY)

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (Harvard):
Lehre, PK & Yao, X 2014, 'Runtime analysis of the (1+1) EA on computing unique input output sequences', *Information Sciences*, vol. 259, pp. 510-531. https://doi.org/10.1016/j.ins.2010.01.031

Link to publication on Research at Birmingham portal

Publisher Rights Statement:
Eligibility for repository : checked 03/06/2014

General rights
Unless a licence is specified above, all rights (including copyright and moral rights) in this document are retained by the authors and/or the copyright holders. The express permission of the copyright holder must be obtained for any use of this material other than for purposes permitted by law.

•Users may freely distribute the URL that is used to identify this publication.
•Users may download and/or print one copy of the publication from the University of Birmingham research portal for the purpose of private study or non-commercial research.
•User may use extracts from the document in line with the concept of 'fair dealing' under the Copyright, Designs and Patents Act 1988 (?)
•Users may not further distribute the material nor use it for the purposes of commercial gain.

Where a licence is displayed above, please note the terms and conditions of the licence govern your use of this document.

When citing, please reference the published version.

Take down policy
While the University of Birmingham exercises care and attention in making items available there are rare occasions when an item has been uploaded in error or has been deemed to be commercially or otherwise sensitive.

If you believe that this is the case for this document, please contact UBIRA@lists.bham.ac.uk providing details and we will remove access to the work immediately and investigate.

Download date: 18. Apr. 2024

Information Sciences 259 (2014) 510–531

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Runtime analysis of the (1 + 1) EA on computing unique input output sequences ☆

Per Kristian Lehre *, Xin Yao [1]

The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK

A R T I C L E   I N F O

Article history:
Received 11 March 2008
Received in revised form 10 November 2008
Accepted 30 January 2010
Available online 21 February 2010

Keywords:
Finite state machines
Conformance testing
Unique input output sequences
Evolutionary algorithms
Random search
Runtime analysis

A B S T R A C T

Computing unique input output (UIO) sequences is a fundamental and hard problem in conformance testing of finite state machines (FSM). Previous experimental research has shown that evolutionary algorithms (EAs) can be applied successfully to find UIOs for some FSMs. However, before EAs can be recommended as a practical technique for computing UIOs, it is necessary to better understand the potential and limitations of these algorithms on this problem. In particular, more research is needed in determining for what instance classes of the problem EAs are feasible, and for what instance classes EAs are provably better than random search strategies.

This paper presents rigorous theoretical and numerical analyses of the runtime of the (1 + 1) EA and random search on several selected instance classes of this problem. The theoretical analysis shows firstly, that there are instance classes where the EA is efficient, while random testing fails completely. Secondly, an instance class that is difficult for both random testing and the EA is presented. Finally, a parametrised instance class with tunable difficulty is presented. The numerical study estimates the constants in the asymptotic expressions obtained in the theoretical analysis, and the variability of the runtime. The numerical results fit well with the theoretical results, even for small problem instance sizes. Together, these results provide a first theoretical characterisation of the potential and limitations of the (1 + 1) EA on the problem of computing UIOs.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

As modern software systems grow larger and more complex, there is an increasing need to support the software engineer with tools for automating some of the software engineering tasks. The field of *search based software engineering* (SBSE) approaches this challenge in a novel way by reformulating software engineering problems into optimisation problems. Such a reformulation has allowed the automation of a wide range of software engineering tasks using evolutionary algorithms and other randomised search heuristics [2].

The increasing popularity of SBSE approaches is partly due to the relatively ease with which search heuristics can be adapted to new problem domains. In principle, the only required ingredients in a search based approach is an encoding of candidate solutions and a way of comparing the quality of two candidate solutions. In contrast, developing problem-spe-

☆ A preliminary version of this paper appeared in [1].

* Corresponding author. Tel.: +44 (0) 121 414 3734; fax: +44 0 121 414 2799.
E-mail addresses: P.K.Lehre@cs.bham.ac.uk (P.K. Lehre), X.Yao@cs.bham.ac.uk (X. Yao).
[1] Tel.: +44 (0) 121 414 3734; fax: +44 (0) 121 414 2799.

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2010.01.031

cific algorithms may require deep insight into the problem structure. The development of problem-specific algorithms is further complicated by the fact that many software engineering problems are NP-hard [3].

However, before SBSE approaches can be widely adopted in the industry, some challenges must be addressed. In particular, it is hard to predict whether a search heuristics will be successful on a given optimisation problem. In some cases, search heuristics fail to find any solution to a problem within acceptable time. Such failures can happen for several reasons. Firstly, the search heuristic applied may not be the most appropriate search heuristic among the many search heuristics that have been developed. From a general point of view, the No Free Lunch (NFL) theorem limits the comparative advantage a given search heuristic can have on a wide problem classes [4]. Although the conditions of the NFL theorem do not always hold for practical problems [5], it is reasonable to assume that the effectiveness of a search heuristic depends on how well it is adapted to the problem. Hence, for a given SE problem, one search heuristic may fail, whereas another succeeds. Secondly, a search heuristic may fail if its parameters are not appropriately tuned to the problem. In the case of evolutionary algorithms, it is known that minor changes in parameters like the use of crossover operator [6,7], population size [8,9], diversity mechanisms [10,11], and the mutation-selection balance [12] can have dramatic impacts on the runtime. A third reason for failure is related to the computational intractability of many software engineering problems. Any search heuristic applied to an NP-hard optimisation problem will fail on at least on some instances of the problem, unless some widely held conjectures in computational complexity do not hold.

Unless these causes of failure are not properly addressed, the SBSE techniques will be associated with some degree of unreliability. We therefore argue that studies of search based approaches to a software engineering problem should consider the questions: Which of the many available search heuristic is best suited for the problem? How should the parameter settings be adjusted? Which instances of the problem are tractable for the search heuristic, and which instances are hard? We claim that such questions would be most rigorously answered by a theoretical analysis. However, except for a few studies [1,13–15], all previous research in SBSE has been experimental.

To answer the questions above rigorously, it is necessary to specify more clearly what it means that a search heuristic is successful on a problem. For a given search heuristic and problem class, one can initially ask 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 [16]. 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 search heuristics [16]. However, convergence itself gives very little information about whether a search heuristic is successful in practice, because no limits are put on the amount of resources the algorithm uses. If convergence can be guaranteed within unlimited time, the next question to ask is how much time the search heuristic needs to find the solution. This type of questions falls within the realms of runtime analysis, where one tries to estimate the runtime as a function of the problem instance size. Similar to the case of classical algorithms, one can make the broad distinction between efficient algorithms that find the solution in polynomial time, and inefficient algorithms, that need exponential time to find the solution. Runtime analysis of search heuristics is very challenging, partly because of their randomised nature. Nevertheless, the techniques for analysing search heuristics have improved rapidly over the last decade, to the point where the runtime of search heuristics can now be analysed on classical problems in combinatorial optimisation [17]. We suggest that the theoretical methods that have been developed for analysing the runtime of search heuristics can be applied in theoretical studies in search based software engineering. Some of the possible avenues for this type of research have already been outlined in [18].

To initiate such a theoretical study, we therefore consider the domain of finite state machine (FSM) testing, which is an area with a long history in software engineering [19]. Techniques for testing finite state machines have traditionally been applied to test implementations of communication protocols [20]. However FSM testing techniques have also been applied elsewhere, including traditional software testing domains [21]. Finite state machine testing has also been a popular research area within search based software engineering [22–28,24,29]. All previous research on FSM testing in search based software engineering has been empirical.

We will focus on the specific problem of computing unique input output (UIO) sequences for finite state machines [19]. Unique input output (UIO) sequences are related to conformance testing of finite state machines, which consists of checking whether an implementation machine is equivalent with a specification machine. While one has full information about the specification machine, the implementation machine is given as a black box. To check the implementation machine for faults, one is restricted to input a sequence of symbols and observe the outputs the machine produces. A fundamental problem which one will be faced with when trying to come up with such checking sequences is the *state verification problem*, which is to assess whether the implementation machine starts in a given state [19]. One way of solving the state verification problem is by finding a unique input output sequence (UIO) for that state. A UIO for a state is an input sequence which, when started in this state, causes the FSM to produce an output sequence which is unique for that state.

Computing UIOs is hard. All known algorithms for this problem have exponential runtime with respect to the number of states. Lee and Yannakakis proved that the decision problem of determining whether a given state has a UIO or not is PSPACE-complete, and hence also NP-hard [30]. In the general case, it is therefore unlikely that there will ever be an efficient method for constructing UIOs and one cannot hope to do much better than random search or exhaustive enumeration. The application of evolutionary algorithms or any other randomised search heuristic cannot change this situation. However, the existence of hard instances does not rule out the possibility that there are many interesting instances that can be solved efficiently with the right choice of algorithm. On such "easy" instances, EAs can potentially be more efficient than exhaustive enumeration and random search.

Guo et al. reformulated the problem of computing UIOs into an optimisation problem to which he applied an EA [24]. When comparing this approach with random search for UIOs, it was found that the two approaches have similar performance on a small FSM, while the evolutionary approach outperforms random search on a larger FSM. Derderian et al. presented an alternative evolutionary approach which also allows the specification machine to be partially specified [22]. Their approach was compared with random search on a set of real-world FSMs and on a set of randomly generated FSMs. Again, it was found that the evolutionary approach outperformed random search on large FSMs. Furthermore, the difference in performance increased with the size of the FSM. Although previous experimental research have show that there are instances of the problem where the evolutionary approach is preferable over a simple random search strategy, more research is needed to get a deeper understanding of the potential of EAs for computing UIOs. Such a deeper insight can only be obtained if the experimental research is complemented with theoretical investigations.

Runtime analysis of EAs is difficult. 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 [17], including sorting [31], minimum spanning tree [32] and Eulerian cycle [33].

Our objective with the theoretical investigation is not to propose a new evolutionary approach to computing UIOs, and we do not claim that the evolutionary approach taken here is more efficient than previous approaches. Rather, we would like to consider a sufficiently simple scenario that allows a theoretical characterisation to be undertaken. We would like to analyse whether evolutionary algorithms can outperform random search in this problem domain, and we would also like to understand what types of FSMs the EA can find UIOs efficiently, and for what types of FSMs the problem is hard for the EA.

This paper extends the preliminary conference version of the paper [1] in several ways. The theoretical study has been complemented with an extensive numerical study. The runtime analysis now considers a generalised variant of the $(1+1)$ EA that operates on general input alphabets. Furthermore, the easy FSM problem instance class has been generalised to modulo-n counters over general input alphabets, and the hard FSM problem instance class is generalised to the wider class of sequence detection FSMs.

The rest of this paper is structured as follows. Section 2 recalls the basic definitions for FSMs and UIO sequences. Section 3 describes the evolutionary approach for computing UIO that will be considered in this paper. In particular, the section describes how candidate solutions are encoded, the definition of the fitness function and how the $(1+1)$ Evolutionary Algorithm has been adapted for the problem. In addition, Section 3 defines the expected runtime of an evolutionary algorithm. Section 4 contains the main theoretical runtime results from the paper. Section 4 describes the experimental methodology, and Section 5 presents the experimental results and compares these with the theoretical results obtained in Section 4. The paper is concluded with a discussion and conclusion in Sections 7 and 8.

2. Preliminaries

2.1. Notation

Symbol ϵ denotes the empty string. The length of a string x is denoted $\ell(x)$. Concatenation of strings x and y is denoted $x \cdot y$, and x^i denotes i concatenations of x. Standard notation (e. g., O, Ω and Θ) for asymptotic growth of functions (see, e.g. [34]) is used in the analysis.

2.2. Finite state machines

Definition 1 (*Finite state machine [19]*). A *finite state machine* (FSM) M is a quintuple $M = (I, O, S, \delta, \lambda)$, where I is the set of input symbols, O is the set of output symbols, S is the set of states, $\delta : S \times I \rightarrow S$ is the state transition function and $\lambda : S \times I \rightarrow O$ is the output function.

At any point in time, an FSM M is in exactly one state s in S. When receiving an input a from I, the machine outputs symbol $\lambda(s,a)$ and goes to state $\delta(s,a)$. The domain of the state transition function δ and the output function λ is generalised to non-empty strings over the input alphabet, i.e.

$$\delta(s, a_1a_2\cdots a_n) := \delta(\delta(s, a_1a_2\cdots a_{n-1}), a_n) \quad \text{and}$$
$$\lambda(s, a_1a_2\cdots a_n) := \lambda(s, a_1)\cdot\lambda(\delta(s, a_1), a_2\cdots a_n).$$

Definition 2 (*Unique input output sequence [19]*). A *unique input output sequence* (UIO) for a state s in an FSM M is a string x over the input alphabet of M such that $\lambda(s,x) \neq \lambda(t,x)$ for all states $t, t \neq s$.

Definitions 1 and 2 are illustrated in Fig. 1. An edge (s_i, s_j) labelled i/o defines the transition $\delta(s_i, i) = s_j$ and the output $\lambda(s_i, i) = o$. The input sequence $acac$ is a UIO for state s_1, because only starting from state s_1 will the FSM output the sequence 0001. The single input symbol c is a UIO for state s_2, because only state s_2 has output 1 on input c. This FSM does not have any

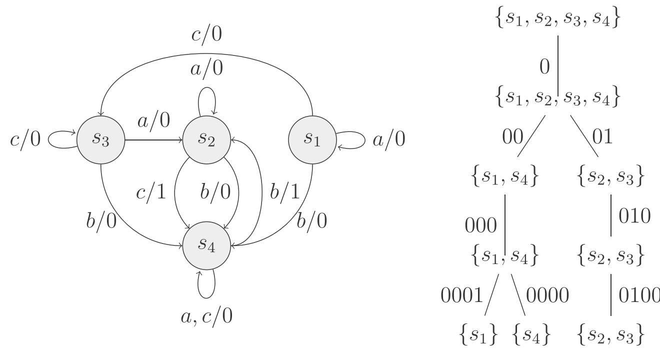

Fig. 1. FSM with corresponding state partition tree for the input sequence *acac*.

distinguishing sequence, because for every input symbol x in the FSM, there exists at least two states s and t such that $\lambda(s,x) = \lambda(t,x)$ and $\delta(s,x) = \delta(t,x)$.

3. UIO generation as an optimisation problem

3.1. Representation and fitness function

Previous research on computing UIOs with EAs have considered different types of representations and fitness functions [22,24]. The purpose of this paper is not to propose a new evolutionary approach to computing UIOs, but to analyse the runtime behaviour of an EA when using what we consider is the most straightforward and simple representation and fitness function. We will therefore build on the existing approaches, and where we consider it natural, make some modifications. The research question as to which representation and fitness function are the most appropriate when computing UIOs is left open for future research.

Following [24], candidate solutions are represented as strings over the input alphabet I of the FSM. The length of the input sequences among which the EA is searching for UIOs has to be bounded in some way. It is known that there exists FSMs where the shortest UIOs have exponential length with respect to the number of states n [30]. However, in order to obtain a fitness function that is computationally feasible, it is necessary to bound the length of input sequences $L(n)$ to some small polynomial in n. As all the FSM classes studied here have UIOs of length n, we therefore bound the input sequence length to $L(n) := n$.

The representation used here differs from previous representations in that we do not use "don't care"-symbols proposed in Guo et al. [24]. Such symbols do not cause any state transition in the FSM, and can therefore be removed from the solutions provided by the EA to obtain UIOs shorter than the representation length $L(n)$. We will not consider "don't care"-symbols because shorter UIOs can still be obtained through a simple post-processing stage. For every UIO x of length $L(n)$ that has been obtained by the EA, it is easy to check in polynomial time whether any prefix $x_1 \cdots x_i$ of x, $1 < i < L(n)$ is also a UIO.

Similarly to Guo et al. [24], we will define the fitness of an input sequence as a function of the *state partition tree* induced by the input sequence. Intuitively, the state partition tree of an input sequence represents how increasingly long prefixes of the input sequence partitions the set of states according to the output they produce. Fig. 1(right) gives an example of a state partition tree for input sequence *acac* on the FSM in Fig. 1(left). The root node is the set of all nodes. On input symbols *ac*, state s_1 and s_4 output 00, while states s_2 and s_3 output symbol 01. The two partitions $\{s_1, s_4\}$ and $\{s_2, s_3\}$, are divided consecutively on further inputs, and finally into three partitions $\{s_1\}, \{s_4\}$ and $\{s_2, s_3\}$ on the final input *acac*. Each singleton $\{s_i\}$ in a state partition tree indicates that the corresponding input sequence is a UIO for that state s_i.

In previous work [22,24] the EA was given the task to obtain UIOs for all states simultaneously. The fitness functions were defined to reward input sequences that are close to be UIOs, regardless of for which state in the FSM. One could hope that given a sufficiently diverse population, different individuals would encode UIOs for different states. However, as was found in previous research [24], it was hard to maintain such a diverse population, and the input sequences tended to converge towards a few similar input sequences. We would argue that the problem of computing UIOs for two states s and t is two different objectives. In general, these objectives can be conflicting, as the UIO for state s can be significantly different from the UIO for state t. As an example, consider the FSM in Fig. 1. The shortest UIO for state s_4 is the sequence b, however none of the other states in this FSM has a UIO beginning with symbol b, hence the objective of computing a UIO for state s_4 conflicts with the objective of computing UIOs for the other states. In the exceptional case that none of the objectives are conflicting, the FSM is likely to contain a distinguishing sequence (DS) [19], and there is no need to apply an EA.

Here, we will therefore consider what we think is a more natural approach, to search for a UIO for one state at a time. UIOs for all the states in the FSM can be obtained simply by re-running the EA, each time with a different fitness function corresponding to a new state. The fitness of an input sequence with respect to a state s is hence defined with respect to the cardinality of the leaf containing state s in the state partition tree.

Definition 3 (*Fitness function*). For an FSM M with n states, define the fitness function $f_{M,s} : I^n \rightarrow \mathbb{R}$ for state s as

$$f_{M,s}(x) := n - \gamma_M(s,x), \quad \text{where}$$
$$\gamma_M(s,x) := |\{t \in S | \lambda(s,x) = \lambda(t,x)\}|.$$

The *instance size* of a fitness function $f_{M,s}$ is defined as the number of states n in FSM M. The value of $\gamma_M(s,x)$ is the number of states in the leaf node of the state partition tree containing node s, and is in the interval from 1 to n. If the shortest UIO for state s in FSM M has length no more than n, then $f_{M,s}$ has an optimum of $n-1$. As an example of Definition 3, consider the FSM in Fig. 1, for which the fitness function takes the values $f_{M,s_1}(acac) = 3$ and $f_{M,s_1}(aaaa) = 0$. In all the instances presented here, the objective is to find a UIO for state s_1. To simplify notation, the notation f_M will therefore be used instead of $f_{M,s}$, and the notation $\gamma(x)$ will be used instead of $\gamma_M(s,x)$, where the FSM M is given by the context.

3.2. Evolutionary algorithms

Runtime analysis of evolutionary algorithms in new problem domains are often initiated with a simple evolutionary algorithm called (1 + 1) EA [35,17]. The (1 + 1) EA keeps a single individual represented as a bitstring of length n, and mutates this individual by flipping each bit with probability $1/n$. One way of directly apply the (1 + 1) EA to the UIO problem would be to encode input symbols in binary. However, such binary encodings are inconvenient for input alphabet cardinalities other than a power of two. Instead, we consider a generalised (1 + 1) EA which operates on any fixed input alphabet I. In this algorithm, the individual is represented as a string of length n over the input alphabet I. In the mutation step, each string position i in a bitstring x is mutated independently with probability $1/n$ by setting x_i to a randomly chosen input symbol r different from the original symbol x_i. Clearly, with the binary input alphabet $I = \{0,1\}$, the generalised (1 + 1) EA becomes identical to the classical (1 + 1) EA.

Algorithm 1. (1 + 1) Evolutionary algorithm.

```
1: Sample x uniformly at random from I^n
2: repeat
3:   x' ← x
4:   for i = 1 to n do
5:     if with probability 1/n then
6:       x'_i ← r, where r is sampled uniformly at random from I \ {x_i}
7:     end if
8:   end for
9:   if f(x') ⩾ f(x) then
10:    x ← x'
11:  end if
12: until termination condition met
```

We say that one *step* of the (1 + 1) EA is one iteration of the *Repeat*-loop in the algorithm. In each step of (1 + 1) EA, the fitness value $f(x')$ must be evaluated. We can assume that the fitness value $f(x)$ of the current search point x is stored in a local variable. Hence, after step t of the algorithm, the fitness function has been evaluated t times. In the *black box scenario*, the runtime complexity of a randomised search heuristic is measured in terms of the number of evaluations of the fitness function, and not in terms of the number of internal operations in the algorithm [36]. For a given function and randomised search heuristic, the *expected runtime* is defined as the mean number of fitness function evaluations until the optimum is evaluated for the first time. The runtime on a class of fitness functions is defined as the supremum of the expected runtimes of the functions in the class [35,37].

A function is considered *easy* for the (1 + 1) EA if the expected runtime is bounded from above by a polynomial in n. Conversely, a function is considered *hard* for the (1 + 1) EA if the expected runtime is bounded from below by an exponential function in n.

4. Runtime analysis

This section analyses the behaviour of the (1 + 1) EA on the problem of computing UIOs for different classes of FSMs. We would like to emphasise that our primary concern here is not with these FSM classes themselves. Rather, the FSM classes are studied to shed light on what distinguishes the tractable from the intractable classes of FSMs for the (1 + 1) EA on the UIO problem. Given the NP-hardness of the UIO problem, it is clear that EAs will fail on certain FSM classes, and we would like to distinguish those from the tractable FSM classes. For this reason, it is desirable to consider FSM classes that are not too intricate or too closely tied to a particular application, such that the reasons for failure or success of the (1 + 1) EA can become

more easily recognisable, and possibly be used as guidelines when studying the behaviour of evolutionary algorithms in more application-near cases of the UIO problem.

Nevertheless, the two first FSM classes that are studied here turn out to play fundamental roles in practical applications of FSMs. The tractable class is the modulo n-counter FSMs [38]. Such FSMs are used widely where there is a need to report when a certain number of events of a given type has occurred. A simple example of such FSMs is the binary counter. The second class of FSMs, which is intractable for the (1 + 1) EA, is the sequence detector FSMs [38]. These FSMs listen to a stream of symbols, and emit a signal every time a specified sequence of symbols occurs in the stream. Sequence detector FSMs are also widely used, for example in the lexical analysis component of compilers, in electronic key locks, or in communication systems where one wants to recognise start and stop signals in a bitstream. Both of these FSM classes typically occur as components within a larger system.

A runtime analysis of the (1 + 1) EA for a given FSM M requires a certain level of information about the fitness landscape of the corresponding fitness function f_M. The runtime analyses are therefore carried out in two steps. The first step is a characterisation of the function values of the fitness function f_M obtained via Definition 3. The second step is the runtime analysis of the search heuristics on fitness function f_M.

As described above, the fitness function is defined with respect to a single state. To obtain UIOs for all n states in an FSM, the EA should be re-run n times, once for each state. The runtime analysis will only focus on the time to find a UIO for a single state s_1 for the following reason. If T_i is the runtime to obtain a UIO for state i, then the overall runtime T to find UIOs for all n states will be $T = \sum_{i=1}^{n} T_i$. Hence, the asymptotic upper and lower bounds on runtime T are $T = O(n \cdot T^*)$ and $T = \Omega(T^*)$, where $T^* := \max_i T_i$. Hence, to characterise within a linear factor the asymptotic runtime to find the UIOs for all states, it suffices to analyse the runtime on the hardest state.

4.1. Easy FSM instance class

Our first aim is to construct a class of problem instances which is hard for random search, while being easy for the (1 + 1) EA. In order to be hard for random search, the length of the shortest UIO for state s_1 must be at least linear in n, and there must be few UIOs of this length. To keep the instance class easy for the (1 + 1) EA, the idea is to ensure that the resulting fitness function has few interactions among the variables. It is well known that the (1 + 1) EA optimises all linear functions efficiently [37,35].

Definition 4 (*Modulo-n counter FSM class*). For instance sizes $n, n \geqslant 2$, define an FSM E with input alphabet I of constant cardinality m having a special input symbol $1 \in I$, and output alphabet $O := \{\texttt{const}, \texttt{incr}, \texttt{reset}\}$, and n states $S := \{s_1, s_2, \ldots, s_n\}$. For all states s_i, define the output function λ as

$$\lambda(s_i, x) := \begin{cases} \texttt{reset} & \text{if } x = 1, \text{ and } i = n, \\ \texttt{incr} & \text{if } x = 1, \text{ and } i < n, \text{ and} \\ \texttt{const} & \text{otherwise.} \end{cases}$$

For all states s_i, define the state transition function δ as

$$\delta(s_i, x) := \begin{cases} s_1 & \text{if } x = 1, \text{ and } i = n, \\ s_{i+1} & \text{if } x = 1, \text{ and } i < n, \text{ and} \\ s_i & \text{otherwise.} \end{cases}$$

The objective is to find an UIO of length n for state s_1.

The instances in Definition 4 are illustrated in Fig. 2. This FSM is a *modulo-n counter*, which counts the number of 1-symbols received, and outputs the special symbol reset after n inputs of symbol 1. Note that $(s_n, s_1, 1/\texttt{reset})$ is the only state transition with a distinguishing input/output behaviour. Furthermore, the states will never collapse, i. e. $\delta(s_i, x) \neq \delta(s_j, x)$ for any input sequence x and any pair of different states s_i and s_j. It is easy to see that any sequence of length n where at most one symbol is different from 1, is a UIO for state s_1. We show that the easy instance class leads to a fitness function which is very similar to the well-known fitness function ONEMAX [39]. The proof is in the appendix.

Proposition 1. *Define the function* $|\cdot|_1 : I \to \{0, 1\}$ *as* $|1|_1 := 1$ *and* $|p|_1 := 0$ *for all* $p \neq 1$. *The fitness function* f_E *corresponding to the instance class in* Definition 4 *takes the following values*

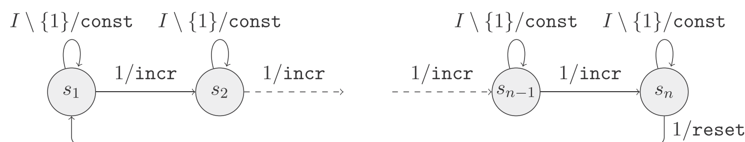

Fig. 2. Finding UIOs for state s_1 is easy with the (1 + 1) EA.

$$f_E(x) = \begin{cases} n-1 & \text{if } x = 1^n, \text{and} \\ \sum_{i=1}^{n} |x_i|_1 & \text{otherwise.} \end{cases}$$

Theorem 1. *Using fitness function f_E, the expected time until random search finds a UIO for state s_1 is geometrically distributed with parameter $p = (nm+1) \cdot m^{-n}$. Furthermore, the probability that random search finds a UIO for state s_1 in less than $e^{c \cdot n}$ iterations is exponentially small $e^{-\Omega(n)}$, where c is a constant.*

Proof. An optimal solution has at most one symbol different from 1. Hence, the probability that the uniformly sampled sequence in any iteration of random search is optimal is $p := (nm+1) \cdot m^{-n}$.

For $n > 5$ and $m \geqslant 2$, we have $p < e^{-n/4}$. The probability that random search finds an optimal solution within $e^{cn}, n \geqslant 6$, steps is thus no more than

$$\sum_{i=1}^{\exp(c \cdot n)} (1 - (nm+1) \cdot m^{-n})^i \cdot (nm+1) \cdot m^{-n} \leqslant \sum_{i=1}^{\exp(c \cdot n)} (nm+1) \cdot m^{-n} \leqslant e^{c \cdot n} \cdot e^{-n/4} = e^{-\Omega(n)},$$

when $c < 1/4$. □

The runtime analysis of (1 + 1) EA on the problem of computing a UIO for state s_1 is similar to the well-known analysis of the (1 + 1) EA on the ONEMAX problem [39]. However, because we consider a generalised search space I^n, where $|I| = m$, we need to consider a more general case.

Theorem 2. *Using fitness function f_E, (1 + 1) EA will find a UIO for the state s_1 in the modulo n counter FSM in expected time $O(nm \log n)$, where $m \geqslant 2$ is the size of the input symbol alphabet of the FSM.*

Proof. By the values of fitness function f_E, in non-optimal search points x and y, $f_E(x) \geqslant f_E(y)$ if and only if search point x has at least as many 1-symbols as search point y. So in a given step of (1 + 1) EA, the mutated search point x' will only be accepted if it has at least as many 1-symbols as search point x. If x' has more 1-symbols than x, we say that the step is successful. When x has i symbols different from 1, the probability of a successful step is at least $i/(n(m-1)) \cdot (1-1/n)^{n-1} \geqslant i/enm$.

Search points with at least $n-1$ 1-symbols are optimal, hence it suffices to wait for $n-1$ successful steps to find the optimum. The expected runtime of (1 + 1) EA is therefore bounded from above by $\sum_{i=2}^{n} enm/i = O(nm \log n)$. □

The result in Theorem 1 means that random search is a highly inefficient strategy for computing UIOs for modulo-n counter FSMs. As such FSMs are quite common in applications, this result means that one cannot rely on random search as a general strategy for computing UIOs. The problem with random search occurs when the shortest UIOs are at least linear in length with respect to the number of states.

In contrast, the result in Theorem 2 implies that the (1 + 1) EA can compute UIOs efficiently for the modulo-n counter FSMs. Although the time to find the UIO increases with the size of the input alphabet, the runtime remains polynomial as long as the input alphabet is of polynomial size. The efficiency of the (1 + 1) EA can be attributed to the structure of the fitness landscape induced by the FSM class. For every input sequence that is not a UIO for state s_1, it suffices to increase the number of 1-symbols in the input sequence to distinguish s_1 from one more state. Hence, at any non-optimal point in the search space, there is a better, neighbouring input sequence.

It has previously been asserted that evolutionary algorithms can outperform random search when computing UIOs. However, Theorems 1 and 2 provide the first formal proof that this assertion is indeed true. This result warrants further exploration of search based approaches for computing UIOs.

4.2. Hard FSM instance class

As explained above, the problem of computing UIOs is NP-hard. Hence, one should expect that any search based approach will fail to find UIOs in polynomial time for at least some classes of FSMs. In practical applications of search heuristics, it is necessary to know about these intractable cases such that other techniques or problem reformulations can be considered to avoid the unnecessary waste of resources. Here, we describe a broad class of FSMs that is hard for both random search and the (1 + 1) EA.

Definition 5 (*Sequence detector FSM class*). Given an alphabet Σ of constant size m, and a word w of length $n-1 > 2$ over Σ, define an FSM H_w with input alphabet $I := \Sigma \cup \{\texttt{EOL}\}$ and output alphabet $O := \{\texttt{ack}, \texttt{found}\}$, and n states $S := \{s_1, s_2, \ldots, s_n\}$. For all states s_i, define the output function λ as

$$\lambda(s_i, x) := \begin{cases} \texttt{found} & \text{if } i = n, \text{ and } x = \texttt{EOL}, \text{ and,} \\ \texttt{ack} & \text{otherwise.} \end{cases}$$

For all states s_i, define the state transition function δ as

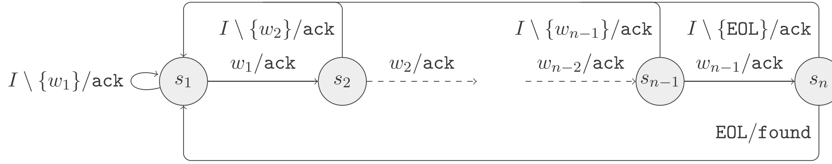

Fig. 3. Finding a UIO for state s_1 is hard for (1 + 1) EA.

$$\delta(s_i, x) := \begin{cases} s_{i+1} & \text{if } i < n, \text{ and } x = w_i, \\ s_1 & \text{otherwise.} \end{cases}$$

The objective is to find an UIO of length n for state s_1.

The instances in Definition 5 are illustrated in Fig. 3, and corresponds to a machine that recognises a fixed keyword w. In every step, the FSM receives a new letter from a word. The FSMs acknowledges each input letter by outputting the symbol ack. If the FSM is given all the input letters corresponding to the keyword w suffixed by a special input symbol EOL marking the "end of line", then the FSM responds with the message found.

The difficulty of computing a UIO for state s_1 in the different members of the Sequence Detector FSM class may vary depending on the particular keyword w that the FSM detects. However, the runtime on a class of functions depends on the hardest instance. In order to prove a lower bound on this class, it therefore suffices to show that there exists at least one sub-class of sequence detector FSMs for which it is hard to compute UIOs. The particular class of FSMs that we will consider have keyword alphabet $\Sigma = \{1\}$ and keyword $w = 1^{n-1}$.

Proposition 2 shows that this instance class leads to a fitness function that takes the same low value on all, except two input sequences. Hence, the fitness landscape is essentially a "needle in the haystack" which is hard for all EAs [36]. The proof is in the appendix.

Proposition 2. *The fitness function f_w corresponding to the instance class in Definition 5 with keyword alphabet $\Sigma = \{1\}$ and keyword $w = 1^{n-1}$, takes the value $f_H(x) = 1$ for all input sequences $x \in I^n$, except on input sequences 1^n and $1^{n-1} \cdot \mathtt{EOL}$ on which it takes the values $f_H(1^n) = 0$ and $f_H(1^{n-1} \cdot \mathtt{EOL}) = n - 1$.*

By noting that the shortest UIO for state s_1 in the special type of keyword FSM instance considered above has length n, the following theorem can be proved similarly to Theorem 1.

Theorem 3. *The Sequence Detector FSM class contains instances such that the probability that random search will find a UIO for state s_1 in less than $e^{c \cdot n}$ iterations is exponentially small $e^{-\Omega(n)}$, where c is a small constant.*

To lower bound the expected runtime of the (1 + 1) EA, we apply drift analysis which is a general technique for proving exponential lower bounds on first hitting-time in Markov processes [37]. The following variant of the drift theorem is taken from [40].

Lemma 1 (Drift theorem). *Let $X_0, X_1, X_2, \ldots$ be a Markov process over a set of states S, and $g : S \to \mathbb{R}_0^+$ a function that assigns to every state a non-negative real number. Pick two real numbers $a(n)$ and $b(n)$ which depend on a parameter $n \in \mathbb{R}^+$ such that $0 < a(n) < b(n)$ holds and let random variable T denote the earliest point in time $t \geqslant 0$ where $g(X_t) \leqslant a(n)$ holds. If there are constants $\lambda > 0$ and $D \geqslant 1$ and a polynomial $p(n)$ taking only positive values, for which the following four conditions hold*

1. $g(X_0) \geqslant b(n)$,
2. $b(n) - a(n) = \Omega(n)$,
3. $E[e^{-\lambda(g(X_{t+1}) - g(X_t))} | X_t, a(n) < g(X_t) < b(n)] \leqslant 1 - \frac{1}{p(n)}$, *for all* $t \geqslant 0$,
4. $E[e^{-\lambda(g(X_{t+1}) - b(n))} | X_t, b(n) \leqslant g(X_t)] \leqslant D$, *for all* $t \geqslant 0$,

then for all time bounds $B \geqslant 0$, the following upper bound on probability holds for random variable T

$$\Pr[T \leqslant B] \leqslant e^{\lambda(a(n) - b(n))} \cdot B \cdot D \cdot p(n).$$

Theorem 4. *The keyword recogniser FSM class contains instances where the probability that (1 + 1) EA will find a UIO for state s_1 in this instance within $e^{c \cdot n}$ steps is exponentially small $e^{-\Omega(n)}$, where c is a small constant.*

Proof. For any n, we consider the Keyword recogniser FSM with keyword alphabet $\Sigma = \{1\}$ and keyword $w = 1^{n-1}$. We lower bound the time it takes until the current search point of (1 + 1) EA contains at least $n - 1$ 1-symbols for the first time. This time is clearly shorter than the time the algorithm needs to find the optimal search point $1^{n-1} \cdot \mathtt{EOL}$.

Let random variables $Y_0, Y_1, Y_2, \ldots$ represent the stochastic behaviour of (1 + 1) EA on fitness function f_H, where each variable Y_t denotes the number of EOL-symbols in the search point in step t. Then $Y_0, Y_1, Y_2, \ldots$ is a Markov process.

To simplify this Markov process, we introduce another Markov process $X_0, X_1, X_2, \ldots$, defined for all $t \geqslant 0$ as $X_0 := Y_0$, and

$$X_{t+1} := \begin{cases} X_t + 1 & \text{when } Y_{t+1} \geqslant Y_t + 2, \quad \text{and} \\ X_t + Y_{t+1} - Y_t & \text{otherwise}. \end{cases}$$

Let random variable T denote the first point in time t where $X_t \leqslant 1$. Intuitively, the simplified process corresponds to an "improved" algorithm which never looses more than one EOL-symbol in each step, but otherwise behaves as the (1 + 1) EA. Clearly, the expected optimisation time $E[T]$ of the modified process is no more than the expected optimisation time of the original process.

The drift theorem is now applied to derive an exponential lower bound on random variable T. Define $g(x) := x$ and parameters $a(n) := 1$ and $b(n) := cn$, where c is a constant that will be determined later. With this setting of $a(n)$ and $b(n)$, the *second condition* of the drift theorem is satisfied.

The following notation will be used

$$p_j := \Pr[g(X_{t+1}) - g(X_t) = j | X_t, 1 < g(X_t) < cn],$$
$$r_j := \Pr[g(X_{t+1}) - g(X_t) = j | X_t, cn \leqslant g(X_t)].$$

The terms in the equation

$$E[e^{-\lambda(g(X_{t+1}) - g(X_t))} | X_t, 1 < g(X_t) < cn] = \sum_{j=-cn}^{n-cn} p_j \cdot e^{-\lambda j} \tag{1}$$

can be divided into four parts according to the value of the index variable j. The term where $j = 1$ simplifies to $p_1 \cdot e^{-\lambda} \leqslant e^{-\lambda}$, the term where $j = 0$ simplifies to $p_0 \cdot e^0 = (1 - 1/n)^n \leqslant 1/e$, the term where $j = -1$ simplifies to

$$p_{-1} \cdot e^{\lambda} \leqslant e^{\lambda}\left(1 - \frac{1}{n}\right)^{n-1} \frac{1}{n} \cdot X_t \leqslant e^{\lambda} c$$

and the remaining terms where $j \leqslant -2$ can be simplified as follows:

$$\sum_{j=2}^{cn} e^{j\lambda} \cdot p_{-j} = \sum_{j=2}^{cn} e^{j\lambda} \frac{1}{n^j}\left(1 - \frac{1}{n}\right)^{n-j}\binom{X_t}{j} \leqslant \sum_{j=2}^{cn} e^{j\lambda} \frac{1}{n^j} \frac{(cn)^j}{j!} = \sum_{j=2}^{cn} \frac{(e^{\lambda}c)^j}{j!} \leqslant -1 - e^{\lambda}c + \sum_{j=0}^{\infty} \frac{(e^{\lambda}c)^j}{j!} = -1 - e^{\lambda}c + \exp(e^{\lambda}c).$$

The sum in Eq. (1) can now be bounded from above as

$$E[e^{-\lambda(g(X_{t+1}) - g(X_t))} | X_t, 1 < g(X_t) < cn] \leqslant e^{-\lambda} + 1/e + e^{\lambda}c - 1 - e^{\lambda}c - \exp(e^{\lambda}c) = e^{-\lambda} + 1/e - 1 + \exp(e^{\lambda}c).$$

For appropriate values of λ and c (eg. $\lambda = \ln 2$ and $c = 1/32$), the value of this expression is less than $1 - \delta$ for a constant $\delta > 0$. Hence, the *third condition* in the drift theorem is satisfied. It is straightforward to see that the *fourth condition* holds now that condition three holds

$$E[e^{-\lambda(g(X_{t+1}) - cn)} | X_t, cn \leqslant g(X_t)] \leqslant E[e^{-\lambda(g(X_{t+1}) - g(X_t))} | X_t, cn \leqslant g(X_t)] = r_1 \cdot e^{-\lambda} + \sum_{j=0}^{n} r_{-j} \cdot e^{j\lambda}.$$

Using the same ideas as above, the expectation can be bounded from above by

$$E[e^{-\lambda(g(X_{t+1}) - cn)} | X_t, cn \leqslant g(X_t)] \leqslant e^{-\lambda} + \exp(e^{\lambda}).$$

When parameter $c = 1/32$, using Chernoff bounds [41], the probability that the first search point has less than cn EOL-symbols is $e^{-\Omega(n)}$. Hence we can assume with high probability that the *first condition* is satisfied as well.

All four conditions of the Drift theorem now hold. By setting $B = e^{c'n}$ for some small constant c', one obtains the exponential lower bound $\Pr[T \leqslant e^{c'n}] = e^{-\Omega(n)}$. □

Theorems 3 and 4 mean that for some sequence detector FSMs, the probability that either random search or (1 + 1) EA find a UIO is very small, even when the search heuristics are allowed an exponential number of iterations in the number of states. Hence, these theorems imply that these search heuristics cannot be applied to find UIOs for such classes of FSMs with the existing representation and fitness function.

For designers of new search based approaches to computing UIOs, Theorem 4 points out an important class of FSMs which could serve as a benchmark for further studies. Given the reasons for failure described in the proofs above, one could seek novel fitness functions or representations which do not suffer from the same problems.

4.3. k-Gap FSM instance class

The previous two subsections presented classes of FSMs for which it is either easy or hard to compute a UIO with the (1 + 1) EA. It is desirable to also study FSM classes of intermediate difficulty. Trivially, one could consider the modulo-n counter as an FSM class of intermediate difficulty, because as Theorem 2 shows, the runtime on this class increases with the size of the input alphabet I. However, we would like to understand cases where it is the structure of the FSM, rather than the size

of the input alphabet that determines the difficulty. We therefore consider a third class of FSMs that has binary input alphabet $I = \{0, 1\}$, and where the structure is parameterised by the value of some parameter k. It will be shown that the instance class is easy when the value of parameter k is low, and the problem becomes harder when parameter k is increased.

Definition 6 (*k-Gap FSM instance class*). For instance sizes $n \geqslant 7$, let k be a constant integer, $2 \leqslant k \leqslant (n-3)/2$ and define $m := n - k - 1$. Define an FSM $G(k)$ with input and output symbols $I := \{0, 1\}$ and $O := \{a, b\}$ respectively, and n states $S := \{s_1\} \cup \{r_1, r_2, \ldots, r_k\} \cup \{q_1, q_2, \ldots, q_m\}$. For all states t in S, define the output function λ as

$$\lambda(t,0) := b \text{ and } \lambda(t,1) := \begin{cases} b & \text{if } t = q_m, \text{and} \\ a & \text{otherwise.} \end{cases}$$

For state s_1, define the state transition function δ as

$$\delta(s_1, 0) := s_1 \quad \text{and} \quad \delta(s_1, 1) := r_1.$$

For states $r_i, 1 \leqslant i \leqslant k$, define the state transition function δ as

$$\delta(r_i, 1) := s_1 \quad \text{and} \quad \delta(r_i, 0) := \begin{cases} q_{k+2} & \text{if} \quad i = k, \quad \text{and} \\ r_{i+1} & \text{otherwise.} \end{cases}$$

And finally, for states $q_i, 1 \leqslant i \leqslant m$, define the state transition function δ as

$$\delta(q_i, 0) := s_1 \quad \text{and} \quad \delta(q_i, 1) := \begin{cases} q_1 & \text{if } i = m, \quad \text{and} \\ q_{i+1} & \text{otherwise.} \end{cases}$$

The objective is to find an UIO of length n for state s_1.

One way of creating a problem with tunable difficulty is to make sure that the fitness function contains a "trap" which easily leads the EA into a local optimum at distance k from the global optimum. By increasing the distance k between the local and global optimum, the problem gets harder [35]. The "trap" in the FSM defined in Definition 6 are the m states $q_1, \ldots, q_m$. By producing an input sequence with many leading 1-bits, the (1 + 1) EA easily makes the output from these states different from state s_1. However, as can be seen from Fig. 4, the UIO for state s_1 must contain k 0-bits somewhere in the beginning of the input sequence. The proofs of the following two propositions are in the appendix.

Proposition 3. *Let z be any string with length* $\ell(z) = k + 1$.

$$f_{G(k)}(10^k 1^{n-2k-2} z) = n - 1. \tag{2}$$

In other words, any search point on the form $10^k 1^{n-2k-2} z$ *is a UIO for state* s_1. *Let i be any integer* $0 \leqslant i < n$, *and z any string of length* $n - i - 1$. *If string z does not contain the substring* 1^{n-2k-2}, *then*

$$f_{G(k)}(1^i 0 z) = \min(i, n - k - 1), \quad \text{and} \tag{3}$$

$$\gamma(1^i) = \gamma(1^i 0 z). \tag{4}$$

Proposition 4. *Let i be any integer* $0 \leqslant i \leqslant 2k + 2$, *and z any sequence of length* $\ell(z) = n - i - 1$ *containing the sequence* 1^{n-2k-2}. *If the sequence* $1^i 0 z$ *is not optimal, then* $f_{G(k)}(1^i 0 z) \leqslant 2k + 2$.

Analysing the (1 + 1) EA on the problem is easy if we can assume that the sequence 1^{n-2k-2} never occurs in the suffix. Proposition 5 shows that this assumption holds in most cases. The proof of this proposition is in the appendix.

Definition 7 (*Typical run*). A *typical run* of (1 + 1) EA on $f_{G(k)}$ is a run where the current search point x is never on the form $1^i 0 z$, $0 \leqslant i < 2k + 2$, where z is a sequence of length $\ell(z) = n - i - 1$ containing the sequence 1^{n-2k-2}. A run of (1 + 1) EA on $f_{G(k)}$ is divided into the following three phases.

Phase 1 is defined as the time interval in which the search point has less than $2k + 2$ leading 1-bits. If the current search point during this phase has a suffix containing sequence 1^{n-2k-2}, then we say that we have a *failure*. The event of failure will be denoted $\mathscr{F}$. *Phase 2* is defined as the time interval when the search point has between $2k + 2$ and $n - k - 1$ leading 1-bits. *Phase 3* is defined as the time interval when the search point has at least $n - k - 1$ leading 1-bits, and this phase lasts until the search point is optimal for the first time.

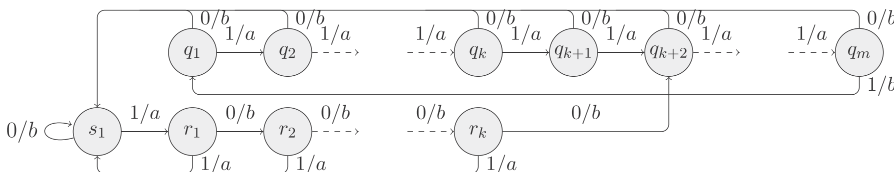

Fig. 4. Finding a UIO for state s_1 with (1 + 1) EA becomes harder when increasing parameter k.

Proposition 5. *The probability of a failure during Phase 1 is bounded from above by* $e^{-\Omega(n)}$.

Theorem 5. *Let k be any constant integer* $k \geqslant 2$. *The expected runtime of (1 + 1) EA to find a UIO for state* s_1 *using* $f_{G(k)}$ *is* $\Theta(n^k)$.

Proof. Given the probability of the failure event $\mathscr{F}$, the expected runtime of (1 + 1) EA can be calculated as

$$E[T] = (1 - \Pr[\mathscr{F}]) \cdot E[T|\overline{\mathscr{F}}] + \Pr[\mathscr{F}] \cdot E[T|\mathscr{F}]. \quad (5)$$

To estimate an *upper bound* on the expected runtime, we use that $E[T] \leqslant E[T|\overline{\mathscr{F}}] + \Pr[\mathscr{F}] \cdot E[T|\mathscr{F}]$.We will first find an upper bound on the runtime conditional on a typical run $E[T|\overline{\mathscr{F}}]$ and pessimistically assume that the optimal search point will not be found during Phase 1 or 2 of the run. We first upper bound the duration of Phase 1 and 2. Let $i, 0 \leqslant i \leqslant n-k-1$, be the number of leading 1-bits in the current search point. A step of the algorithm is called successful if the mutated search point x' has more leading 1-bits than the current search point x. In typical runs, Proposition 3 guarantees that x' will be accepted in a successful step. To reach the end of Phase 2, we have to wait at most for $n-k-1$ successful steps. The probability of a successful step is at least $1/n \cdot (1-1/n)^{n-1} \geqslant 1/en$, so the expected duration of Phase 1 and Phase 2 is $O(n^2)$. By Proposition 3, for Phase 3 to end, it is sufficient to flip k consecutive 1-bits starting at position 2. The probability that this will happen in any step of Phase 3 is at least $(1/n)^k \cdot (1-1/n)^{n-k} \geqslant 1/(n^k e)$. Hence, the expected duration of Phase 3 is bounded from above by $O(n^k)$. An upper bound on the expected runtime conditional on the event that the run is typical is therefore $E[T|\overline{\mathscr{F}}] = O(n^k)$.

We now give an upper bound on the expected time $E[T|\mathscr{F}]$ conditional on a failure. To keep the analysis simple, we give a pessimistic upper bound. At some time in such a run, the current search point has a suffix containing the sequence 1^{n-2k-2}. We assume that this search point is not the optimal search point, and furthermore, we assume that in this situation, we will never accept an optimal search point during Phase 1. Clearly, this will only slow down the optimisation process. By Proposition 4, this search point has fitness at most $2k+2$. To end Phase 1, Proposition 3 shows that it is sufficient to wait for a step in which all the 0-bits in the $2k+3$ long prefix of the search point is flipped into 1-bits. The probability of such a mutation is at least $(1/n)^{2k+3}(1-1/n)^{n-2k-3} \geqslant 1/en^{2k+3}$. So if a failure occurs, the duration of Phase 1 will be no longer than $O(n^{2k+3})$. Failures do not occur in Phases 2 or 3, we therefore reuse the upper bounds of $O(n^2)$ and $O(n^k)$ that were calculated for the typical runs, yielding an upper bound of $O(n^{2k+3})$ for the duration of runs with failures. Due to the exponentially small failure probability, the unconditional expected runtime of (1 + 1) EA is therefore $E[T] = O(n^k)$.

A *lower bound* on the expected runtime is estimated using the inequality $E[T] \geqslant (1 - \Pr[\mathscr{F}]) \cdot E[T|\overline{\mathscr{F}}]$. We need to estimate the expected runtime conditional on a typical run. Optimal search points contain the suffix 1^{n-2k-2}, hence the optimal search point will not be found during Phase 1 of typical runs. By Propositions 3 and 4, only search points with at least $2k+2$ leading 1-bits or an optimal search point will be accepted during Phase 2. Optimal search points must contain $10^k 1^{n-2k-2}$. Hence, in order to find the optimum in the second phase, it is necessary to flip k consecutive 1-bits into 0-bits, starting somewhere in the interval between position 2 and $k+2$. The probability of this event in any given step is no more than k/n^k. Hence, the expected duration of Phase 2 and Phase 3 is at least n^k/k steps. The unconditional expected runtime can now be bounded from below by $E[T] \geqslant (1 - e^{-\Omega(n)}) \cdot n^k/k = \Omega(n^k)$. □

Theorem 5 relates the runtime of the (1 + 1) EA to a structural parameter k in a class of FSMs. The theorem shows that small modifications in the structure of an FSM can have a strong impact on the runtime of the algorithm. Hence, one cannot generally infer that the runtime of (1 + 1) EA will be similar for similarly structured FSMs.

The reason why the runtime of the (1 + 1) EA increases with parameter k can be explained informally as follows. In order for state s_1 to reach the distinguishing transition from state q_m to q_1, it is necessary to include k consecutive 0-symbols early in the input sequence. However, on most input sequences containing early 0-symbols, the output when starting in state s_1 will be indistinguishable from the outputs when starting in most of the states labelled q. The majority of the states are q-states. In contrast state s_1 is distinguished from q-states on input sequences with many leading 1-bits. When the (1 + 1) EA has found an input sequence with sufficiently many leading 1-bits, it is necessary to flip k 1-bits simultaneously to produce a better input sequence. The expected waiting time for such an event is n^k.

5. Numerical studies

5.1. Objectives

The theoretical analysis in the previous section leaves some questions open. One open question is how the search heuristics behave on particular, possibly small, instance sizes. The theoretical analysis of runtime considered asymptotic behaviour and the results were expressed using big-Oh notation. Hence, in principle, there is no guarantee that these theoretical results are relevant for "interesting" values of n, *i.e.* for the number of states in "typical" applications of the FSMs considered. A statement like $f(n) = O(g(n))$ only means that there exists some constants c and n_0 such that $f(n) < c \cdot g(n)$ for all $n > n_0$. If either of these constants are very large, the inequality is either very weak, or only holds for very large values of n. Two questions to consider are therefore whether the sizes occurring in runtime expressions are very large, and whether the runtime results seem to manifest themselves for small values of n.

A second issue is that the theoretical analysis mainly focused on the expectation of the runtime distribution. To get a clearer picture of the runtime distributions, it is necessary to investigate the variability of the distributions.

5.2. Strategy

To address these issues, the theoretical analysis is complemented with a numerical study. The modulo-n counter FSMs include a wide range of different FSMs that depends both on the number of states n and the input alphabet I. Similarly, the sequence detector FSMs contains a wide range of FSMs depending on the keyword alphabet Σ, the keyword w, and the number of states n. To keep the number of experiments within a feasible range, we therefore only consider the modulo-n counter FSMs with the binary input alphabet $I = \{0, 1\}$, and we will refer to these FSMs as the Easy FSM instance class. Furthermore, we only consider the sequence detector FSMs with keyword alphabet $\Sigma = \{1\}$, and keyword $w = 1^{n-1}$. These FSMs correspond exactly to those FSMs considered in the proof of Theorem 4. They will in the following be referred to as the Hard FSM instance class. Additionally, the runtime of (1 + 1) EA will be investigated on the k-gap FSM instance class for values of k between 2 and 4. Based on the theoretical results on the expected runtime of (1 + 1) EA on this instance, it is deemed impractical to carry out experiments for values of k higher than 4.

For each experimental setting, the search heuristic under consideration is started, and run until an optimal solution has been found. The runtime of one run is defined as the number of times the fitness function has been evaluated. Repeated runs with different random seeds are made to allow statistical analysis. Table 1 summarises the experimental settings. The experiments are numbered from 1 to 14, and divided into two groups.

The purpose of the first group of experiments numbered 1–7 is to study the relationship between instance size and runtime for different instance classes, and in particular to provide estimates for the constants in the asymptotic runtime expressions. The theoretical runtime analysis combined with estimates of the time to evaluate the fitness function were used to choose sets of instance sizes which were deemed feasible to carry out within reasonable time constraints.

For each setting of algorithm, FSM instance and instance size n, 100 experiments were run. The bootstrap percentile method with 400 bootstrap samples were used to calculate 95% confidence intervals of the mean of the true runtime distribution. These confidence intervals are reported as error bars. Following [42], 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 $a \cdot t(n)$ of the runtime, where the model parameter a corresponds to the constant to be estimated. The residual sum of squares (RSS) of each fitted model was calculated to identify the model which corresponds best with the observed runtimes. Additionally, for each instance size, a box-and-whisker plot is made, indicating the smallest observed runtime, the lower quartile, the median, the upper quartile and the largest observed runtime. Together, the box-and-whisker plots provide information on how the variability of runtime depends on the instance size.

The purpose of the second group of experiments, numbered 8–14, is to look closer at the variability of the runtime for a fixed instance size. In these experiments, a larger number of repetitions were made. Again, the particular instance sizes were chosen based on the theoretical analysis and estimates of the time to evaluate the fitness function. The results from these experiments are plotted as histograms to visualise the variability of the observed runtime distributions.

6. Numerical results

6.1. Estimating constants

6.1.1. Easy FSM instance class

The easy FSM instance class was constructed to point out that there are instance classes where (1 + 1) EA is highly efficient, whereas random search fails completely. The theoretical analysis shows that the expected runtime of (1 + 1) EA on this

Table 1
Summary of parameter settings in numerical experiments.

	Algorithm	Instance class	Instance sizes n	Repetitions
1	(1 + 1) EA	Easy FSM	$n \in \{10, 60, 110, \dots, 710, 760\}$	100
2	RS	Easy FSM	$n \in \{4, 5, 6, \dots, 25, 26\}$	100
3	(1 + 1) EA	Hard FSM	$n \in \{4, 5, 6, \dots, 19, 20\}$	100
4	RS	Hard FSM	$n \in \{4, 5, 6, \dots, 19, 20\}$	100
5	(1 + 1) EA	k-gap FSM, $k = 2$	$n \in \{10, 15, 20, \dots, 45, 50\}$	100
6	(1 + 1) EA	k-gap FSM, $k = 3$	$n \in \{10, 15, 20, \dots, 45, 50\}$	100
7	(1 + 1) EA	k-gap FSM, $k = 4$	$n \in \{15, 20, \dots, 45, 50\}$	100
8	(1 + 1) EA	Easy FSM	$n = 200$	2000
9	RS	Easy FSM	$n = 17$	2000
10	(1 + 1) EA	Hard FSM	$n = 13$	2000
11	RS	Hard FSM	$n = 13$	2000
12	(1 + 1) EA	k-gap FSM, $k = 2$	$n = 20$	2000
13	(1 + 1) EA	k-gap FSM, $k = 3$	$n = 20$	2000
14	(1 + 1) EA	k-gap FSM, $k = 4$	$n = 20$	2000

instance class is $O(n \log n)$, and there is an exponentially small probability that random search will find the optimal solution within e^{cn} iterations, where c is some constant.

Three models were fitted to the observed runtimes of (1 + 1) EA on the Easy FSM instance class. The models were chosen to be close to the theoretically obtained runtime bound on this instance. The results summarised in Table 2 indicate that the model which fits the data best is $a \cdot n \log n$ with estimated parameter $a = 1.967$. This result corresponds well with the asymptotic runtime $O(n \log n)$ obtained in Theorem 2. The fitted models are plotted together with the mean of the observed runtimes in Fig. 5.

As show in Fig. 6, the runtime of random search on the easy FSM instance class grows much faster than the runtime of (1 + 1) EA. The log-plot indicates that the runtime grows exponentially with the number of states in the FSM.

6.1.2. Hard FSM instance class

The Hard FSM instance class was constructed to point out that (1 + 1) EA is not successful on all instance classes of this problem. The theoretical analysis shows that on this instance class, there is an exponentially small probability that the algo-

Table 2
Residual sum of squares of three models fitted to the observed runtime of (1 + 1) EA on the Easy FSM instance.

Model	Fit	RSS
$a \cdot n$	12.35	1.411e+09
$a \cdot n \log n$	1.967	1.254e+09
$a \cdot n^2$	0.0209	2.843e+09

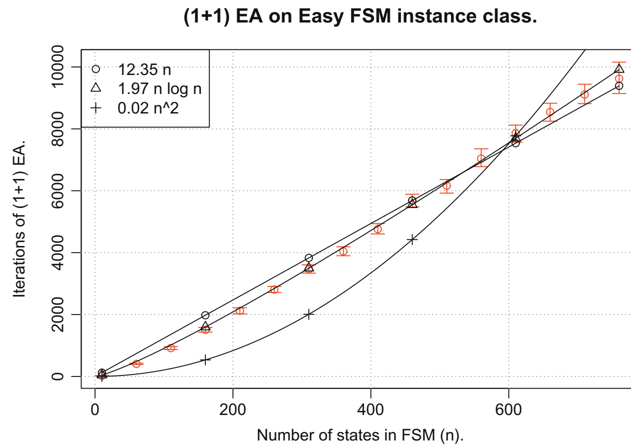

Fig. 5. Mean runtime of (1 + 1) EA on the Easy FSM instance class with 95% confidence intervals plotted with error bars. The fitted models, as given by the legend, are plotted.

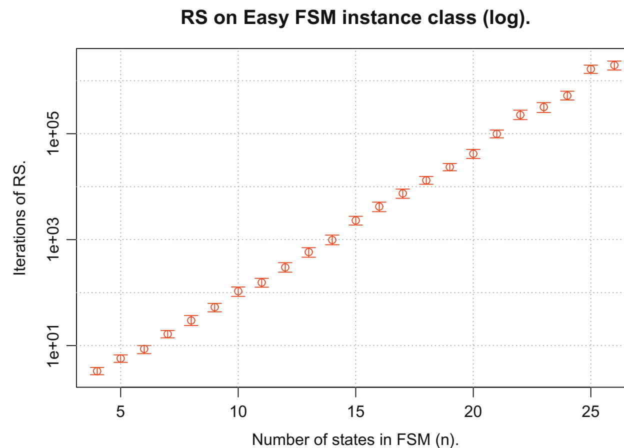

Fig. 6. Mean runtime of random search on the Easy FSM instance class with error bars indicating 95% confidence intervals. The y-axis is logarithmically scaled.

rithm will find the optimal solution within e^{cn} iterations, for some constant c. This result also implies that the expected runtime of (1 + 1) EA on this instance class is exponential.

The plot in Fig. 7 shows the mean runtime of (1 + 1) EA on the Hard FSM instance class with error bars indicating 95% confidence intervals. The log-plot indicates that the observed mean runtime grows exponentially with the number of states in the FSM. The mean observed runtime of Random Search on the same instance class is plotted in Fig. 8, showing a similar trend. These results correspond well with the theoretical results.

6.1.3. k-Gap FSM instance class

The k-gap FSM instance class was constructed to point out that the asymptotic runtime of (1 + 1) EA on the UIO problem is not limited to being either very small or exponentially large, but can range over a large range of values depending on characteristics of the FSMs. Even small changes to the FSM can have a big impact on the runtime. The theoretical analysis shows that the expected runtime of (1 + 1) EA on the k-gap FSM instance class is $\Theta(n^k)$ for any constant $k \geqslant 2$.

Three models were fitted to the observed runtimes of (1 + 1) EA on the k-gap FSM instance class with $k = 2, k = 3$ and $k = 4$, using non-linear regression. The results are summarised in Tables 3–5. For $k = 2$, the model with best fit was $a \cdot n^2$ with estimated parameter $a = 2.518$, for $k = 3$, the model with best fit was $a \cdot n^3$ with estimated parameter $a = 1.722$, and for $k = 4$, the model with best fit was $a \cdot n^4$ with estimated parameter $a = 1.605$. These results correspond well with the theoretical result. The fitted models are plotted with the mean observed runtime in Figs. 9–11. The asymptotic behaviour predicted by the theoretical analysis seems evident in these plots even for small instance sizes. The results from bootstrapping confidence intervals of the mean are shown using error bars in the plots. The error bars indicate larger confidence intervals with increasing instance size.

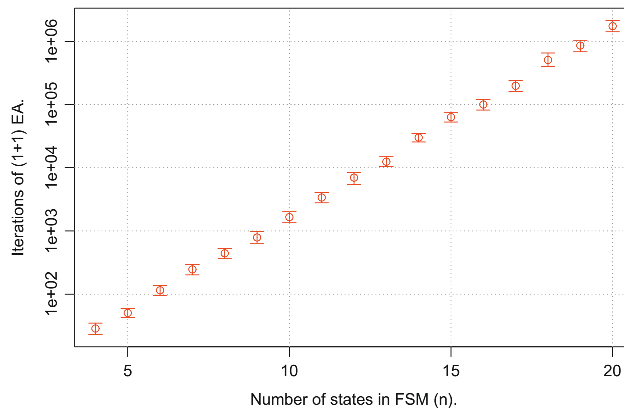

Fig. 7. Mean runtime of (1 + 1) EA on the Hard FSM instance class with error bars indicating 95% confidence intervals. The y-axis is logarithmically scaled.

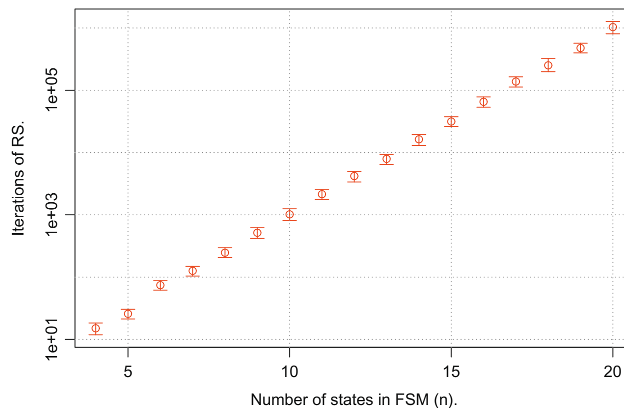

Fig. 8. Mean runtime of random search on the Hard FSM instance class with error bars indicating 95% confidence intervals. The y-axis is logarithmically scaled.

Table 3
Residual sum of squares for (1 + 1) EA on the k-gap FSM instance class ($k = 2$).

Model	Fit	RSS
$a \cdot n^2$	2.518	4.7e+09
$a \cdot n^3$	0.05671	4.789e+09
$a \cdot n^4$	0.001207	5.18e+09

Table 4
Residual sum of squares for (1 + 1) EA on the k-gap FSM instance class ($k = 3$).

Model	Fit	RSS
$a \cdot n^2$	74.67	9.085e+12
$a \cdot n^3$	1.722	8.762e+12
$a \cdot n^4$	0.03725	8.852e+12

Table 5
Residual sum of squares for (1 + 1) EA on the k-gap FSM instance class ($k = 4$).

Model	Fit	RSS
$a \cdot n^2$	3153	1.862e+16
$a \cdot n^3$	73.56	1.758e+16
$a \cdot n^4$	1.605	1.739e+16

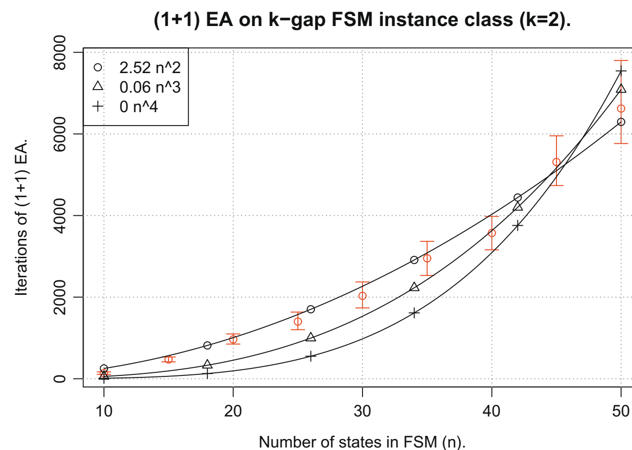

Fig. 9. Mean runtime of (1 + 1) EA on the k-gap FSM instance class ($k = 2$), with error bars indicating 95% confidence intervals. The fitted models, as given by the legend, are plotted.

6.2. Variability of runtime

Box-and-whisker plots are made to show how the variability of the runtime depends on the instance size. The plots in Figs. 12 and 13 show that the interquartile range in the observed runtimes increases with increasing instance size. The increasing interquartile range is most evident in Fig. 13, showing the runtime of Random Search on the Easy FSM instance class.

The box-and-whisker plots for the Hard FSM instance class are not included here, but they show a similar tendency as the box-and-whisker plots for Random Search on the Easy FSM instance class.

The box-and-whisker plots for the k-gap FSM instance class also show that the interquartile range increases with the instance size. Here, we only include the plot for $k = 4$. For larger instance sizes, one can observe that the distribution is positively skewed, with the median closer to the lower than the upper quartile.

To investigate closer the variability in runtime, the experiments numbered 8–14 in Table 1 were conducted with a larger number of repetitions on a single experimental setting. Results from these experiments are summarised in histograms. The histogram in Figs. 14 and 15 shows the observed runtimes from 2000 runs of the (1 + 1) EA on the Easy FSM instance class

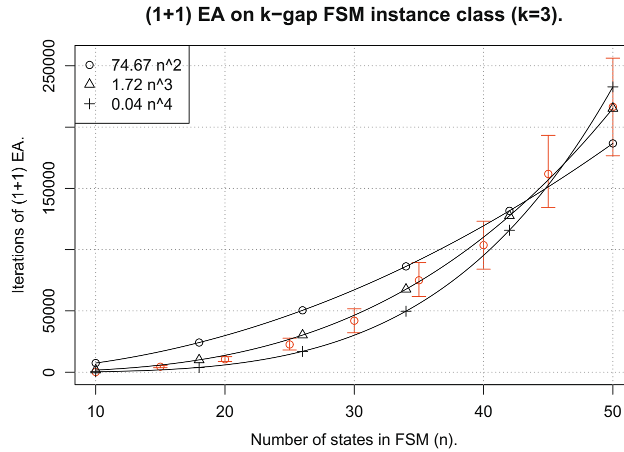

Fig. 10. Mean runtime of $(1 + 1)$ EA on the k-gap FSM instance class $(k = 3)$, with error bars indicating 95% confidence intervals. The fitted models, as given by the legend, are plotted.

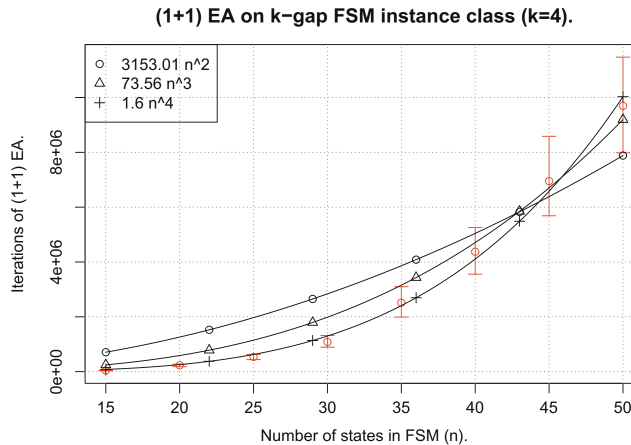

Fig. 11. Mean runtime of $(1 + 1)$ EA on the k-gap FSM instance class $(k = 4)$, with error bars indicating 95% confidence intervals. The fitted models, as given by the legend, are plotted.

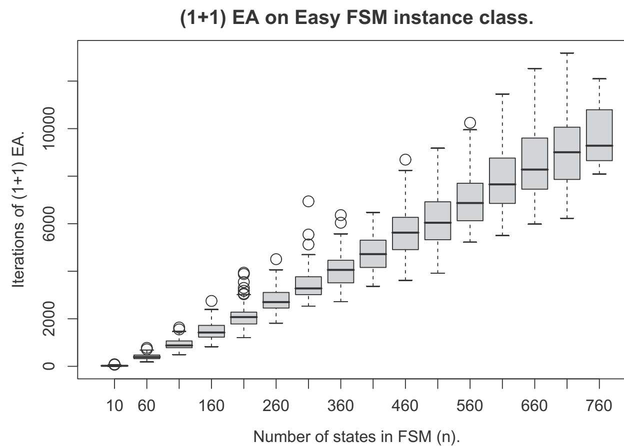

Fig. 12. Box-and-whisker-plots from observed runtimes of $(1 + 1)$ EA on the Easy FSM instance class.

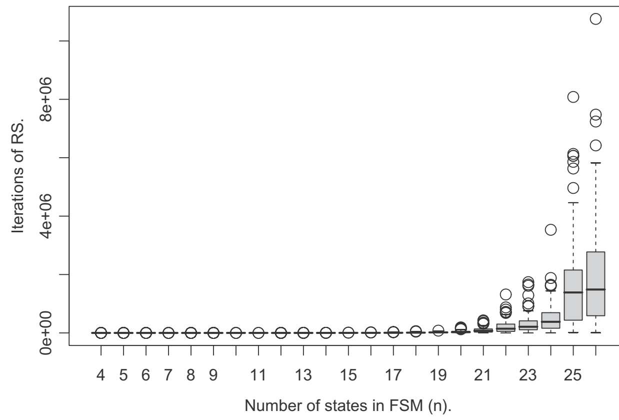

Fig. 13. Box-and-whisker-plots from observed runtimes of random search on the Easy FSM instance ($n = 17$).

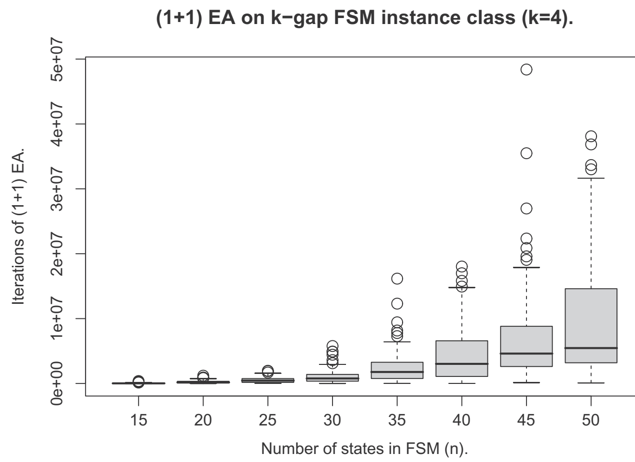

Fig. 14. Box-and-whisker plots of runtime distributions of (1 + 1) EA on the k-gap FSM instance class with $k = 4$.

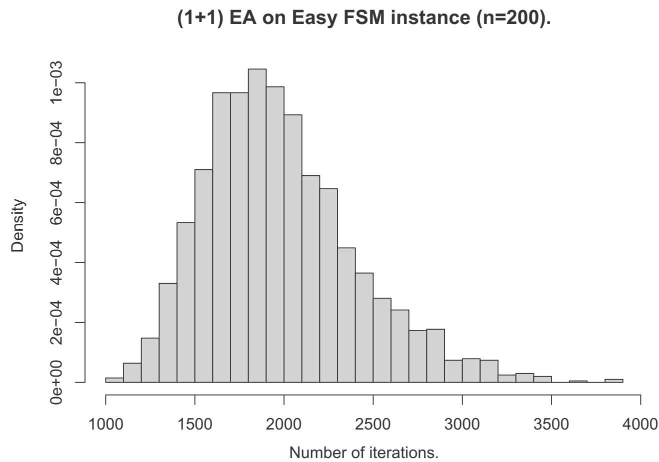

Fig. 15. Histogram of observed runtimes of (1 + 1) EA on the Easy FSM instance class with instance sizes $n = 200$.

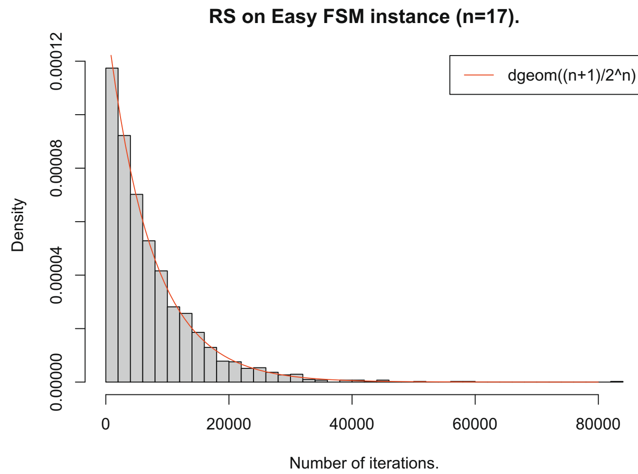

Fig. 16. Histogram of observed runtimes of Random Search on Easy FSM with instance size $n = 17$, with line showing the density function for the geometric distribution with parameter $p = (n + 1) \cdot 2^{-n}$.

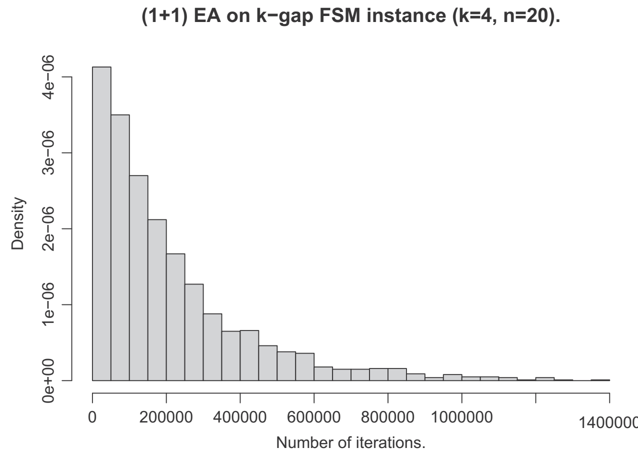

Fig. 17. Histogram of observed runtimes of (1 + 1) EA on the k-gap FSM instance class with instance size $n = 20$ and $k = 4$.

($n = 200$). The histogram shows a slightly positively skewed distribution. The variance of this distribution is lower than the variance of the distribution shown in the histogram in Fig. 16, which summarises the observed runtimes from 2000 runs of Random Search on the same instance class ($n = 17$). The distribution shown by this histogram is highly positively skewed. Theorem 1 shows that the runtime distribution of random search on this instance class is exactly the geometric distribution with parameter $p = (n + 1) \cdot 2^{-n}$. The histogram corresponds well with the plot of the density function of this distribution.

The histograms for the Hard FSM instance class are not included here, but they show similarly shaped distributions as the one of Random search on the Easy FSM instance class in Fig. 16.

The histograms for the runtime of (1 + 1) EA on the k-gap FSM instance class are similar, and we only include the one for $k = 4$ which is shown in Fig. 17. The distribution in this histogram has a similar shape as the distribution of Random Search on the Easy FSM instance. The distribution is highly positively skewed, and has a large variance. In the proof of Theorem 5, it is shown that the dominating phase of a typical run of (1 + 1) EA on this instance class is when the algorithm needs to flip k consecutive bits in a single iteration. Hence, one can conjecture that for large k, the runtime distribution of (1 + 1) EA on the k-gap FSM instance class will be close to geometrically distributed.

7. Discussion

Three classes of finite state machines have been constructed and studied in this paper. The first FSM class is the modulo-n counting FSMs that are used widely in cases where it is necessary to detect when a certain number of events has happened. A simple example is a binary counter. The second FSM class is the sequence detector FSMs, which is also occur frequently in

applications, including in the lexical analysis component of compilers, in communication systems and in electronic key locks. FSMs of these two types often occur as sub-modules within larger systems. The third FSM class, which is parametrised, shares properties with both the easy and hard FSM class. Such FSMs have many applications in software engineering, including modelling and testing of non-functional requirements. One example is security testing of automated teller machines (ATMs). The lock FSM could model requirements related to authentication by personal identification number (PIN), while the counter FSM could model requirements related to card retainment after a specified number of failed authentication attempts.

The notions of easy and hard instances depend on both the EA used and on the way the problem of finding UIOs has been defined. This paper uses the terms hard and easy relative to the (1 + 1) EA, as described in Section 3.2. These terms should not be confused with the terms EA-hard and EA-easy which are sometimes used in evolutionary computation to mean problems that are thought to be generally hard, respectively easy for *all* EAs. There are certainly functions that are hard in the sense of Section 3.2 for (1 + 1) EA, but which are easy for other EAs. Furthermore, the hardness of finding UIOs is relative to the way the fitness function for this problem has been defined. We believe the formulation in Definition 3 is quite natural, however one could envisage other fitness function definitions which could potentially lead to different runtimes for the (1 + 1) EA.

8. Conclusion

Search based software engineering is a promising approach to automating software engineering tasks. Although a significant amount of research has been conducted in the area over recent years, there exists still very few theoretical results. Theoretical research is needed to rigorously determine the potential and limitations of search heuristics in various software engineering domains. In particular, for many software engineering problems that are NP-hard [3], it is necessary to characterise the problem instances that are tractable for search heuristics. Only when the tractable class of problem instances have been accurately characterised can search heuristics be applied with a predictable performance to a software engineering problem.

Here, we have initiated such a theoretical study by analysing the runtime of the (1 + 1) Evolutionary Algorithm on the problem of computing unique input output (UIO) sequences in finite state machines. The primary purpose of this theoretical study has been to give an initial description as to which types of FSMs that are tractable for the (1 + 1) EA, and which classes are intractable. As far as we know, this paper, along with a preliminary conference version [1], represents the first rigorously obtained result on the runtime of an evolutionary algorithm in the field of search based software engineering.

It is shown that on the class of modulo-n counter FSMs, the (1 + 1) EA is highly efficient, whereas random search fails completely. This result indicates that the (1 + 1) EA can be preferable over the sometimes proposed strategy of randomly searching for UIOs. Furthermore, it is shown that the (1 + 1) EA fails on the class of sequence detecting FSMs. On this particular instance class, the state partition tree gives little information about the UIO. The existence of such hard instances for the (1 + 1) EA is to be expected since the general problem of finding UIOs is NP-hard. This result implies that alternative approaches should be considered when computing UIOs for such FSMs. Furthermore, the sequence detecting FSMs could be a useful benchmark for new search based approaches to the UIO problem. Finally, an instance class with tunable difficulty for the (1 + 1) EA is presented. This instance class highlights how specific, small changes to the structure of the FSM can make the problem of computing UIOs increasingly hard. This result implies that structurally similar FSMs are not necessary equally hard for the (1 + 1) EA.

The theoretical analysis was complemented with an extensive numerical study to investigate the constants that are hidden by the big Oh-expressions, and to gain insight into the variability of the runtime. Constants were estimated using nonlinear regression, and variability were investigated using histograms. The numerical and theoretical results agree well. In all cases, the theoretically obtained asymptotic expression for runtime fitted the observed runtimes best among a selection of similar models. The estimated constants in the asymptotic expressions were small. The asymptotic behaviour predicted by the theoretical analysis appear evident, even for small instance sizes. The observed variability in runtime was in general large on all instance classes. The (1 + 1) EA on the Easy FSM instance class showed the least variability. On the other instance classes, both Random Search and (1 + 1) EA showed a much larger variability in runtime. The observed runtimes formed a highly positively skewed distribution.

The stochastic behaviour of evolutionary algorithms and other search heuristics is often highly complex and therefore hard to predict. Only recently have results about the runtime of EAs started to appear for artificial functions and some combinatorial optimisation problems. It is a highly non-trivial task to estimate the success probability of an EA on an arbitrary problem instance. More theoretical research is still needed before such predictions can be made for any given class of FSMs on the UIO problem. However, we think that this and other theoretical studies will contribute to building the strong foundation that is needed for search based approaches to be reliably applied in the software engineering industry.

Acknowledgements

The authors would like to thank Mark Harman, Rob Hierons, Simon Poulding, Rami Bahsoon, Pietro Oliveto, Ramón Sagarna and Andrea Arcuri for their useful comments. This work was supported by EPSRC under Grant No. EP/D052785/1.

Appendix A. Proofs

Proof of Proposition 1. The case where $\sum_{i=1}^{n}|x_i|_1 \geqslant n-1$ is easy. State s_1 is the only state which outputs a on each of the first $n-1$ inputs of symbol 1. Hence, for such input sequences, $\gamma(x) = 1$.

Before showing that the proposition also holds for the remaining input sequences, we first show that for any input sequence x with $\gamma(x) > 1$, we have

$$\gamma(x) = \gamma(x \cdot p) + |p|_1. \tag{A.1}$$

Eq. (A.1) obviously holds when symbol p is different from 1, because all states output symbol `const` on input symbols different from 1, so it remains to show that $\gamma(x \cdot 1) = \gamma(x) - 1$ for all x.

By the definition of the transition function, there must be a state t such that $\delta(t, x) = s_n$. Furthermore, we can show that state s_1 and state t produce the same output on input sequence x. Suppose not, that $\lambda(s_1, x) \neq \lambda(t, x)$. This would imply that on input x, state t must have reached the only distinguishing transition from state s_n to state s_1, i. e. sequence x can be expressed on the form $x = y1z$ with $\delta(t, y) = s_n$. Since both $\delta(t, y)$ and $\delta(t, y1z)$ equal state s_n, we must have $\sum_{i=1}^{\ell(z)}|z_i|_1 \geqslant n-1$. However, this is a contradiction, because the assumption $\gamma(x) > 1$ implies that $\sum_{i=1}^{n}|x_i|_1 < n-1$. It is thus clear that $\lambda(s_1, x) = \lambda(t, x)$, and furthermore $\lambda(s_1, x \cdot 1) \neq \lambda(t, x \cdot 1)$. For all other states s_i different than state t, $\lambda(\delta(s_i, x), 1) = \lambda(\delta(s_1, x), 1) = a$. So to conclude, if $\gamma(x) > 1$ then $\gamma(x) = \gamma(x \cdot 1) + 1$.

We can now show that the proposition also holds for input sequences where $\sum_{i=1}^{n}|x_i|_1 < n-1$. On such input sequences, state s_2 cannot reach the distinguishing state transition from s_n to s_1. So state s_1 and s_2 are indistinguishable and $\gamma(x) > 1$. Obviously, the same also holds for all prefixes of input sequence x. Eq. (A.1) can now be applied recursively, and by noting the special case of $\gamma(\epsilon) = n$ on the empty string, we obtain the desired result.

$$\begin{aligned}
\gamma(x_1 \cdots x_n) &= \gamma(x_1 \cdots x_{n-2}x_{n-1}) - |x_n|_1 \\
&= \gamma(x_1 \cdots x_{n-2}) - |x_{n-1}|_1 - |x_n|_1 \\
&\;\;\vdots \\
&= n - \sum_{i=1}^{n}|x_i|_1. \qquad \square
\end{aligned}$$

Proof of Proposition 2. The two special cases 1^n and $1^{n-1} \cdot \mathtt{EOL}$ are simple. By the definition of the output function, $\lambda(s_i, 1^n) = \mathtt{ack}^{\,n}$ for any state s_i. Hence, $\gamma(1^n) = n$ so the value of the fitness function on the first special case is $f_H(1^n) = 0$. On input sequence $1^{n-1} \cdot \mathtt{EOL}$, the output function gives $\lambda(s_1, 1^{n-1} \cdot \mathtt{EOL}) = \mathtt{ack}^{\,n-1} \cdot \mathtt{found}$, and for states s_i different than s_1, the output function gives $\lambda(s_i, 1^{n-1} \cdot \mathtt{EOL}) = \mathtt{ack}^{\,n}$. Hence, the value of the fitness function on the second special case is $f_H(1^{n-1} \cdot \mathtt{EOL}) = n-1$.

The remaining input sequences to consider are those that contain at least one EOL-symbol, but which are different from the sequence $1^{n-1} \cdot \mathtt{EOL}$. Such strings are of the form $1^k \cdot \mathtt{EOL} \cdot z$ where k is an integer, $0 \leqslant k < n-1$, and z can be any sequence of length $\ell(z) = n-k-1$.

We claim that for any state s_i and such sequences z, if $\lambda(s_1, 1^k \cdot \mathtt{EOL}) = \lambda(s_i, 1^k \cdot \mathtt{EOL})$, then $\lambda(s_1, 1^k \cdot \mathtt{EOL} \cdot z) = \lambda(s_i, 1^k \cdot \mathtt{EOL} \cdot z)$. Suppose otherwise, that $\lambda(s_1, 1^k \cdot \mathtt{EOL}) = \lambda(s_i, 1^k \cdot \mathtt{EOL})$ but $\lambda(s_1, 1^k \cdot \mathtt{EOL} \cdot z) \neq \lambda(s_i, 1^k \cdot \mathtt{EOL} \cdot z)$. But then we must have $\lambda(\delta(s_1, 1^k \cdot \mathtt{EOL}), z) \neq \lambda(\delta(s_i, 1^k \cdot \mathtt{EOL}), z)$, which implies the contradiction that $\lambda(s_1, z) \neq \lambda(s_1, z)$.

We now show that for the sequences on the form $1^k \cdot \mathtt{EOL} \cdot z$, there is exactly one state s_i for which $\lambda(s_1, 1^k \cdot \mathtt{EOL} \cdot z) \neq \lambda(s_i, 1^k \cdot \mathtt{EOL} \cdot z)$. We have just proved that this inequality requires that $\lambda(s_1, 1^k \cdot \mathtt{EOL}) \neq \lambda(s_i, 1^k \cdot \mathtt{EOL})$. Because all states have the same output on input 1, it is necessary that $\lambda(\delta(s_1, 1^k), \mathtt{EOL}) \neq \lambda(\delta(s_i, 1^k), \mathtt{EOL})$, which implies that $\lambda(s_{1+k}, \mathtt{EOL}) \neq \lambda(s_{i+k}, \mathtt{EOL})$. The only way to satisfy this inequality is to let $i + k = n$. Hence, state s_{n-k} is the only state that produces different output than state s_1 on input sequences containing at least one EOL-symbol, and that are different from $1^{n-1} \cdot \mathtt{EOL}$. $\square$

Proof of Proposition 3. We first prove Eq. (2). On input sequence 10^k, only $\delta(s_1, 10^k) = q_{k+2}$ and for all other states t, $\delta(t, 10^k) = s_1$. Hence, state s_1 goes through the distinguishing transition on input 10^k1^{n-2k-2} while all other states are in transition between states s_1 and r_1, showing that state s_1 has a unique output. Therefore, search points on the form $10^k1^{n-2k-2}z$ are optimal. (There are other optimal search points, but knowing the structure of a few optimal search points will be sufficient in the analysis.)

We now show that $\gamma(1^i0) = n - \min(i, m)$. Note that $(q_m, q_1, 1/b)$ is the only distinguishing state transition. For i no larger than m, the i states $q_{m-i+1}, \ldots, q_m$ reach this transition on input sequence 1^i and therefore produce different outputs than state s_1. For i at least m, all m states $q_1, \ldots, q_m$ reach the distinguishing transition. State s_1 and the k states $r_1, \ldots, r_k$ do not reach the distinguishing transition on input sequence 1^i0. Therefore, the number of states that produce different outputs than state s_1 on input sequence 1^i0 is $\min(i, m)$.

Finally, we prove Eqs. (3) and (4) under the assumption that z does not contain the substring 1^{n-2k-2}. For all states s, either $\delta(s, 1^i0) = s_1$ or $\delta(s, 1^i0) = r_2$. All state transition paths from either state s_1 or state r_2 to the distinguishing state transition from state q_m must go through the $n - 2k - 2$ state transitions between q_{k+2} and q_m. Transitions along this path require an input sequence with $n - 2k - 2$ consecutive 1-bits, which is not possible with sequence z. Therefore, we have $\gamma(1^i0z) = \gamma(1^i0) = n - \min(i, m)$. This also proves Eq. (4) because $\gamma(1^i) = \gamma(1^i0) = n - \min(i, m)$. □

Proof of Proposition 4. Assume first that $i = 0$, i. e. the search point begins with a 0-bit. In this case, all states $q_1, \ldots, q_m$ collapse with state s_1, and the suffix z can at most distinguish s_1 from the k states $r_1, \ldots, r_k$. Hence, in this case $\gamma(0z) \geqslant n - k$.

Assume now that $1 \leqslant i \leqslant k + 2$. After input 1^i0, all states have moved to either state s_1 or state r_2. If i is even, then state s_1 has collapsed with states $q_1, \ldots, q_m$. Hence, the suffix z can at most distinguish the k states $r_1, \ldots, r_k$ from state s_1, i. e. $\gamma(1^i0z) \geqslant n - i - k \geqslant n - (k + 2) - k$. If i is odd, then states $r_1, \ldots, r_k$ have collapsed with states $q_1, \ldots, q_m$. So if x is not optimal, then $\gamma(1^i0z) = n - i \geqslant n - k - 2$.

Finally, assume that $k + 2 < i \leqslant 2k + 2$. After input 1^i0, no more states can reach the distinguishing transition because moving from state s_1 or state r_2 to the distinguishing transition requires at least the subsequence $0^{k-1}1^{n-2k-2}$ which is longer than subsequence z. So in this case, we have $\gamma(1^i0z) = n - i \geqslant n - (2k - 2)$. □

Proof of Proposition 5. The current search point x of (1 + 1) EA in Phase 1 is on the form $x = 1^i0z$ for some i, $0 \leqslant i < 2k + 2$ and z a string of length $\ell(z) = n - i - 1$. We call this substring z occurring after the first 0-bit the *suffix* of the current search point.

We first show that as long as the run for the first t steps has been typical, then the suffix z in step $t + 1$ is a random string. The initial search point is a random string, so the suffix is also a random string. Assume that the run has been typical until step t and the suffix z is a random string. By Eq. (4) in Proposition 3, any bitflip of the suffix will be accepted. Randomly mutating a random string, will clearly produce a new random string. The suffix in step $t + 1$ will therefore be a random string. The suffix z of the new search point in step $t + 1$ can contain 1^{n-2k-2}, i. e. we may have a failure in step $t + 1$. However, we show that this is unlikely. The probability that the string 1^{n-2k-2} occurs in a random string shorter than n is no more than $(2k + 2) \cdot 2^{-n+2k+2}$, which for large n is less than $e^{-n/16}$. One way of increasing the number of leading 1-bits without having a failure is by flipping the first 0-bit and flip no other bits. So the probability of increasing the number of leading 1-bits without having a failure in the following step is at least $(1/n) \cdot (1 - 1/n)^{n-1} \geqslant 1/en$.

Hence, for large n, the probability that the number of leading 1-bits increases before we have a failure is at least

$$\frac{1/en}{1/en + 1/e^{n/16}} \geqslant 1 - ne \cdot e^{-n/16} \geqslant 1 - e^{-n/32}.$$

A failure must occur before the number of leading 1-bits has been increased more than $2k + 2$ times. So the failure probability $\Pr[\mathscr{F}]$ is no more than

$$\sum_{i=0}^{2k+2} (1 - e^{-n/32})^i \cdot e^{-n/32} \leqslant (2k + 2) \cdot e^{-n/32} = e^{-\Omega(n)}. \qquad \square$$

References

[1] P.K. Lehre, X. Yao, Runtime analysis of (1 + 1) EA on computing unique input output sequences, in: Proceedings of 2007 IEEE Congress on Evolutionary Computation (Cec'2007), IEEE Press, 2007, pp. 1882–1889.

[2] J. Clarke, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, K. Rees, M. Roper, M.J. Shepperd, Reformulating software engineering as a search problem, Iee Proceedings-Software 150 (3) (2003) 161–175.

[3] M. Harman, The current state and future of search based software engineering, in: Proceedings of International Conference on Software Engineering/ Future of Software Engineering 2007 (Icse/Fose 2007), IEEE Computer Society, 2007, pp. 342–357.

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

[5] S. Droste, T. Jansen, I. Wegener, Optimization with randomized search heuristics – the (a)nfl theorem, realistic scenarios, and difficult functions, Theoretical Computer Science 287 (1) (2002) 131–144.

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

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

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

[9] O. Giel, P.K. Lehre, On the effect of populations in evolutionary multi-objective optimization, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (Gecco'2006), Acm, New York, NY, USA, 2006, pp. 651–658.

[10] T. Friedrich, P.S. Oliveto, D. Sudholt, C. Witt, Theoretical analysis of diversity mechanisms for global exploration, in: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (Gecco'2008), ACM, New York, NY, USA, 2008, pp. 945–952.

[11] T. Friedrich, N. Hebbinghaus, F. Neumann, Rigorous analyses of simple diversity mechanisms, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (Gecco'2007), 2007, pp. 1219–1225.

[12] P.K. Lehre, X. Yao, On the impact of the mutation-selection balance on the runtime of evolutionary algorithms, in: Proceedings of the Tenth Acm Sigevo Workshop on Foundations of Genetic Algorithms (Foga'2009), ACM, New York, NY, USA, 2009, pp. 47–58.

[13] M. Harman, P. McMinn, A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation, in: Proceedings of the Issta 2007 Symposium, 2007, pp. 73–84.

[14] A. Arcuri, P.K. Lehre, X. Yao, Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem, in: Icstw '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, IEEE Computer Society, Washington, DC, USA, 2008, pp. 161–169.

[15] A. Arcuri, Theoretical analysis of local search in software testing, in: Proceedings of Stochastic Algorithms: Foundations and Applications (Saga'2009), 2009, pp. 156–168.
[16] G. Rudolph, Finite markov chain results in evolutionary computation: a tour d'horizon, Fundamenta Informaticae 35 (1) (1998) 67–89.
[17] P.S. Oliveto, J. He, X. Yao, Time complexity of evolutionary algorithms for combinatorial optimization: a decade of results, International Journal of Automation and Computing 4 (1) (2007) 100–106.
[18] P.K. Lehre, X. Yao, Runtime analysis of search heuristics on software engineering problems, Frontiers of Computer Science in China 3 (1) (2009) 64–72.
[19] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines – a survey, Proceedings of the IEEE 84 (8) (1996) 1090–1123.
[20] D. Sidhu, T.-K. Leung, Formal methods for protocol testing: a detailed study, IEEE Transactions on Software Engineering 15 (4) (1989) 413–426.
[21] G.V. Bochmann, A. Petrenko, Protocol testing: review of methods and relevance for software testing, in: Issta '94: Proceedings of the 1994 ACM Sigsoft International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 1994, pp. 109–124.
[22] K.A. Derderian, R.M. Hierons, M. Harman, Q. Guo, Automated unique input output sequence generation for conformance testing of Fsms, The Computer Journal 49 (3) (2006) 331–344.
[23] K.A. Derderian, R.M. Hierons, M. Harman, Q. Guo, Generating feasible input sequences for extended finite state machines (Efsms) using genetic algorithms, in: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (Gecco'2005), ACM Press, New York, NY, USA, 2005, pp. 1081–1082.
[24] Q. Guo, R.M. Hierons, M. Harman, 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), LNCS, vol. 2931, 2004, pp. 164–177.
[25] K.A. Derderian, Automated test sequence generation for finite state machines using genetic algorithms, Ph.D. thesis, Brunel University, 2006.
[26] Q. Guo, R.M. Hierons, M. Harman, K.A. Derderian, Constructing multiple unique input/output sequences using metaheuristic optimisation techniques, IEEE Proceedings Software 152 (3) (2005) 127–140.
[27] Q. Guo, R.M. Hierons, M. Harman, K.A. Derderian, Heuristics for fault diagnosis when testing from finite state machines, Software Testing Verification and Reliability 17 (1) (2007) 41–57.
[28] Q. Guo, R.M. Hierons, M. Harman, K.A. Derderian, Improving test quality using robust unique input/output circuit sequences (uiocs), Information and Software Technology 48 (8) (2006) 696–707.
[29] Q. Guo, Improving fault coverage and minimising the cost of fault identification when testing from finite state machines, Ph.D. thesis, Brunel University, 2006.
[30] D. Lee, M. Yannakakis, Testing finite-state machines: state identification and verification, IEEE Transactions on Computers 43 (3) (1994) 306–320.
[31] J. Scharnow, K. Tinnefeld, I. Wegener, Fitness landscapes based on sorting and shortest paths problems, in: Proceedings of 7th Conf. on Parallel Problem Solving from Nature (Ppsn–Vii), LNCS, vol. 2349, 2002, pp. 54–63.
[32] F. Neumann, I. Wegener, Randomized local search, evolutionary algorithms, and the minimum spanning tree problem, Theoretical Computer Science 378 (1) (2007) 32–40.
[33] B. Doerr, C. Klein, T. Storch, Faster evolutionary algorithms by superior graph representation, in: Proceedings of the 1st IEEE Symposium on Foundations of Computational Intelligence (Foci'2007), 2007, pp. 245–250.
[34] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., McGraw Hill, New York, NY, 2001.
[35] S. Droste, T. Jansen, I. Wegener, On the analysis of the (1 + 1) Evolutionary Algorithm, Theoretical Computer Science 276 (2002) 51–81.
[36] S. Droste, A rigorous analysis of the compact genetic algorithm for linear functions, Natural Computing 5 (3) (2006) 257–283.
[37] J. He, X. Yao, A study of drift analysis for estimating computation time of evolutionary algorithms, Natural Computing 3 (1) (2004) 21–35.
[38] Z. Kohavi, Switching and Finite Automata Theory, second ed., McGraw-Hill, 1978.
[39] H. Mühlenbein, How genetic algorithms really work I. Mutation and hillclimbing, in: Proceedings of the Parallel Problem Solving from Nature 2, (PPSN-II), Elsevier, 1992, pp. 15–26.
[40] O. Giel, Zur analyse von randomisierten suchheuristiken und online-heuristiken, Ph.D. thesis, Universität Dortmund, 2005.
[41] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
[42] T. Jansen, On the brittleness of evolutionary algorithms, in: Foundations of Genetic Algorithms, Lecture Notes in Computer Science, vol. 4436, Springer, Berlin/Heidelberg, 2007, pp. 54–69.