Identifying Worst-Case User Scenarios for Performance Testing of Web Applications Using Markov-Chain Workload Models

Ahmad, Tanwir; Truscan, Dragos; Porres Paltor, Ivan

Published in:
Future Generation Computer Systems

DOI:
10.1016/j.future.2018.01.042

Published: 01/01/2018

Identifying Worst-Case User Scenarios for Performance Testing of Web Applications Using Markov-Chain Workload Models

Tanwir Ahmad*, Dragos Truscan, Ivan Porres

Faculty of Science and Engineering
Åbo Akademi University
Vattenborgsvägen 5, 20500 ÅBO, Finland

Abstract

The poor performance of web-based systems can negatively impact the profitability and reputation of the companies that rely on them. Finding those user scenarios which can significantly degrade the performance of a web application is very important in order to take necessary countermeasures, for instance, allocating additional resources. Furthermore, one would like to understand how the system under test performs under increased workload triggered by the worst-case user scenarios. In our previous work, we have formalized the expected behavior of the users of web applications by using probabilistic workload models and we have shown how to use such models to generate load against the system under test. As an extension, in this article, we suggest a performance space exploration approach for inferring the worst-case user scenario in a given workload model which has the potential to create the highest resource utilization on the system under test with respect to a given resource. We propose two alternative methods: one which identifies the exact worst-case user scenario of the given workload model, but it does not scale up for models with a large number of loops, and one which provides an approximate solution which, in turn, is more suitable for models with a large number of loops. We conduct several

*Corresponding author
Email addresses: tanwir.ahmad@abo.fi (Tanwir Ahmad), dragos.truscan@abo.fi (Dragos Truscan), ivan.porres@abo.fi (Ivan Porres)

experiments to show that the identified user scenarios do provide in practice an increased resource utilization on the system under test when compared to the original models.

Keywords: Performance testing, Markov Chain, Genetic algorithms, Graph-search algorithms

1. Introduction

A tremendous growth has been seen in the field of web technologies during the last two decades. The role of the web applications has changed from the traditional document presentation system to the feature-rich distributed application that is accessible worldwide. Web applications are increasingly being utilized by a large number of companies to run critical business tasks. Thus, ensuring the reliable and stable performance of web applications is imperative for these companies. Poor performance makes the end-users abandon the use of web applications and can cause reputational and financial damage to those companies which rely on web-based platforms [1].

Performance testing is the process of evaluating the responsiveness and scalability of a system under test (SUT) when it is under a certain *synthetic workload* [2] corresponding to a specified number of concurrent *virtual users*. During this process, different *key performance indicators* (KPIs) (e.g., CPU, memory utilization) are monitored in order to determine the performance level of the SUT.

In order to raise the level of abstraction and to promote the reuse and faster update of performance test specifications, in our previous work, we have investigated how the expected behavior of the users of web applications can be specified by using probabilistic models [3, 4]. Such models capture the expected behavior of a set of users by encoding information about the order of their interactions with the SUT, the delay (*think time*) between these interactions, and the probability of a given sequence of interactions to occur. Each traversal of the model graph simulates a timed sequence of interactions between the virtual

user and the web application. By simulating concurrently one model for each virtual user, we can generate the corresponding synthetic workload using our MBPeT model-based performance testing tool [3].

In many situations (e.g., stress testing) one would like to know, before the performance testing session begins, the worst-case user scenario in a given workload model that will potentially trigger the highest utilization of a given resource on the SUT. Such scenario can then be used to benchmark the SUT for possible performance bottlenecks. In practice, this implies finding the path in the workload model graph which will generate the sequence of interactions with the highest resource utilization on the SUT over a sustained period of time.

In this article, we attempt to answer two research questions:

1. RQ1: how can we identify the sequence of interactions in a given workload model which causes the highest utilization of a given resource of the SUT under a sustained period of time?

2. RQ2: what is the scalability of the proposed approach?

In order to answer RQ1, we propose two distinct methods for identifying the worst-case user scenario. The first method is based on graph-search algorithms and provides the exact solution, whereas the second method provides a near-optimal solution. For validation purposes, we run an example where the solutions resulting from applying the two proposed methods are used to generate synthetic workload against the SUT. We then compare the resource utilization they trigger on the SUT with the one triggered by the original workload model.

In order to answer RQ2, we analyze and compare the two methods with respect to their complexity and, respectively, to the precision of the solution, and discuss their benefits and drawbacks.

The rest of the paper is structured as follows: Section 2 provides background information on our proposed approach. We describe the process of load generation for performance testing in Section 3. Section 4 presents an overview of the related work. In Section 5, we describe in detail the steps of our approach. We

empirically validate and evaluate our approach in Section 6 , while we present conclusions in Section 7.

2. Using Markov Chains for Modeling the Workload

A Discrete Time Markov Chain (DTMC) [5] is a discrete time stochastic process which has the property that given the current state, the future of the process is conditionally independent of the previous states. Let $X_n, n = 0, 1, 2, 3$ be a stochastic process which takes on a finite number of states or values which can be written as a set of non-negative integers $\{0, 1, 2, \dots\}$. If the process is currently in state s_n at time n, we denote it as $X_n = s_n$. If we suppose that whenever the process is in state s_n, the process will change its state to s_{n+1} with a fixed probability $P_{s_n s_{n+1}}$, then we can state the property of Markov chains as

$$P\{X_{n+1} = s_{n+1} | X_n = s_n, X_{n-1} = s_{n-1}, \dots X_1 = s_1, X_0 = s_0\} = P_{s_n s_{n+1}}$$

where $P_{s_n s_{n+1}}$ denotes the probability of transitioning from one state to another state in a single step (or one unit of time), and it is known as the *one-step transition probability.*

We model the expected behavior of the users using a slightly modified version of DTMC model, which we formally define as a tuple $M = (S, T, U^r, Util^r, s_I)$ where:

1. $S = \{s_0, s_1, \dots, s_n\}$ is a finite set of states;
2. $T = \{t_0, t_1, \dots, t_n\}$ is a finite set of transitions, such that $t_i = \{\langle s_i, s_j \rangle | s_i, s_j \in S\}$ for all i, j $0 \leq i, j \leq n$;
3. $U^r = \{u_0^r, u_1^r, ..., u_n^r\}$ is a finite set of resource utilizations for a given resource r, and u_i^r is in correspondence with s_i. $Util^r$ is a mapping function from s_i to u_i^r so that $Util^r(s_i) = u_i^r$;
4. $s_I \in S$ is the start state.

Informally, we extend DTMC with two additional labels on the edges: probability value and think time. The *probability value* specifies the chances of that particular edge being chosen according to a probability mass function, whereas the *think time* represents the amount of time that a user waits between two consecutive interactions. In addition, each state in the DTMC model is tagged with an *action* specifying the interaction between the user and the SUT. An action specifies either an HTTP request or a set of HTTP requests that the user sends to the SUT whenever the corresponding state is visited.

The DTMC model in Figure 1 shows a workload model of an auctioning web application, which allows registered users to search, browse, and bid on auctions that other users have created. For instance, after performing a *browse()*, the user can execute either *get_auctions()* action with a probability of 0.87 (after waiting for 3 seconds) or *exit()* with a probability of 0.03 after waiting for 2 seconds. In the model, *start()* and *exit()* are pseudo-states which are only used to indicate the initial and the optional final state of the model, respectively, and they cause no interaction with the SUT.

Different works suggested that such workload models can be obtained from either the requirements of the SUT or Service Level Agreements (SLAs) [4], or by analyzing the historical usage of the system [6] [7] [8]. The workload model in Figure 1 is built using the latter approach following the method described in [6].

3. Workload generation

In this paper, we use our MBPeT (Model-based Performance Testing) [3] tool for load generation. MBPeT is an online performance testing tool, which generates a synthetic semi-random workload against the SUT by simulating a workload model, such as the one in Figure 1 for each concurrent virtual user. The simulation of a workload model for a virtual user begins from the *start()* state. On each state, the tool chooses the next state according to the probability mass function of the current state, while observing the think time values on the

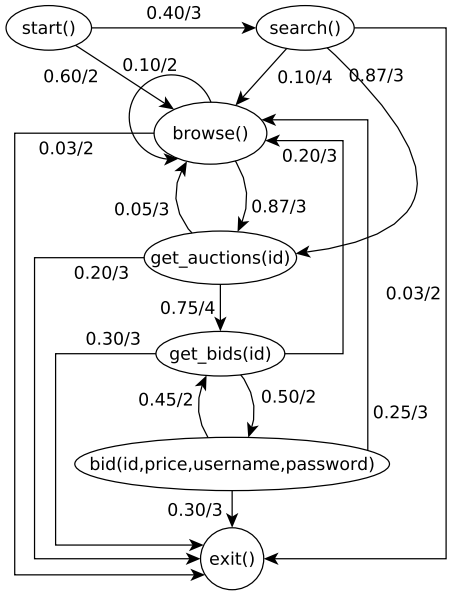

Figure 1: Markov Chain model of a user

visited edges. The simulation ends when the *exit()* state is reached. In short, a sequence of states is generated and executed during the simulation. Whenever a state is visited, the corresponding action is executed against the SUT via a test adapter.

There are different parameters of the testing process that can be provided as command line parameters to the tool such as a ramp function (specifying the amount of concurrent virtual users during a test session), duration of the test session, etc. The workload is generated in a distributed fashion, using multiple load generating nodes, and applied in real-time to the SUT, while measuring several key performance indicators, such as response time, throughput, error rate, etc. At the end of the test session, a detailed test report is provided.

4. Related work

A large number of previous studies (e.g., [9, 10, 11, 12]) have used *performance modeling* for predicting and detecting the performance bottlenecks in

web applications. In these studies, the authors estimate the performance of the system at design time using design specifications. In contrast, we evaluate the performance of the system after it has been fully implemented and we use models which describe the expected behavior of the user.

Bogrdi *et al.* [13] modify the Mean-Value Analysis evaluation algorithm to model the behavior of the thread pool. The proposed algorithm is applied to performance prediction of web-based software systems in the ASP.NET environment. It is assumed in the paper that the thread pool attributes are dominant factors when considering the response time and throughput performance metrics of a web application. As opposed to our approach, the approach in [13] can only be applied to specific web applications.

Gao *et al.* [14] model a composite web service using queuing networks for performance analysis and bottleneck identification. A composite web service consists of several service centers and the internal control flow of those service centers is represented by a Markov chain. In contrast, we use the Markov Chain model as a workload model to capture the expected user behavior and, we try to identify performance bottlenecks by finding the worst-case user scenario in the workload model.

Stewart and Shen [15] present a profile-driven performance model for cluster-based multi-component online services. Application profiles are constructed offline to characterize component resource needs and inter-component communications. The model is used to predict the system throughput by identifying and quantifying the performance bottlenecks within different operating environments. The primary objective of the approach is to predict system performance according to a given component placement and replication strategy; however, we aim to find the worst-case user scenario in a given workload model that can significantly degrade the performance of the system.

Hernandez-Orallo and Vila-Carbó [16] propose a histogram-based workload model to describe any class of web traffic distribution. In order to reflect the second-order statistics (long-range dependence and self-similarity) of the workload, this basic model has been extended using the Hurst parameter. The

authors introduce a set of procedures and operations that work with histograms (histogram calculus) to define the web performance model. In this approach, the authors are interested in the arrival rate of the web traffic whereas we focus on the user behavior and nature of the web traffic.

In recent years, the web applications have become very complex and there are many factors to be considered when the performance of those applications is concerned [17]. Thus, traditional performance models, which are built at the design phase during the software development life cycle, are not flexible and comprehensives enough to be used for performance prediction in the complex web systems. It is reported that supercomputers can predict natural phenomena better than the performance of complex systems [18].

To summarize, even though a large amount of research work has been devoted to investigating the methods for predicting the performance of web applications, we could not find any approach similar to the one discussed in this paper, that is using workload (or user behavioral) models, instead of performance models, to predict the performance of web applications. In this paper, on the one hand, we improve our previous work (presented in [19]) with a more efficient heuristic algorithm. On the other hand, we propose an alternative method to compute the exact solution of the worst-case user scenario. In both cases, we take into account the *think time* between different actions when computing the solution, which was not included in our previous work.

5. Identifying the worst-case user scenario

As stated in the introduction, our goal is to identify a particular user scenario in a workload model (as the one in Figure 1) that would create the highest resource utilization on the SUT. In other words, we are looking for that sequence of states that visited repeatedly over a considerable amount of time will cause the highest resource utilization per state. For brevity, we denote this sequence of states as the *worst path*.

We define a path in workload model as a sequence of states of arbitrary

length n starting in the initial state S_I of the workload model, as follows:

Definition 1 (Path): Given a workload model M, a *path* p is a sequence of transitions $\langle t_0, t_1, \ldots, t_m \rangle$, where $0 < m \leq n$, $t_0 = \{\langle s_0, s_1 \rangle | s_0 = S_I\}$, $t_i \neq t_{i+1}$ for all $i, 0 \leq i \leq m$, and m is the length of a path.

Each state in the workload model corresponds to an action executed by the virtual user against the SUT, which will result at runtime in a certain utilization level of the resources of the SUT. Therefore, we define the *resource utilization of a path with respect to a given resource* as the sum of individual resource utilizations triggered on the SUT by each state of that path:

Definition 2 (Resource utilization of a path): Given a path $p = \langle \langle s_0, s_1 \rangle, \langle s_1, s_2 \rangle, \ldots, \langle s_{m-1}, s_m \rangle \rangle$, the *resource utilization* of a path p wrt. a given resource type r is defined as $U_p^r = Util^r(s_0) + Util^r(s_1) + \ldots + Util^r(s_m)$.

Definition 3 (Worst path): Given a workload model M, the *worst path* p_w of the model is the path with the highest resource utilization per state, such that

$$U_{p_w}^r = Max(U_{p_k}^r / |p_k|)$$

where $0 < k \leq N$, N is the total number of paths in the model, and $|p_k|$ represents the length of path p_k.

We address the problem of finding the worst path in a workload model as an optimization problem and we propose two alternative methods to address this: an exact method using graph-search algorithms and an approximate method using genetic algorithms. Both methods require two common preliminary steps, as shown in Figure 2. These steps will be described in the following.

5.1. Perform preliminary benchmarking

In order to compute which path of the model will create the highest resource utilization on the server, we need to know what is the average resource utilization on the server corresponding to each state. For this purpose, we benchmark each action defined in a given workload model with respect to different resource types (e.g., CPU, memory), by executing the model against the SUT for a given period of time. For the approach presented in this article, we record

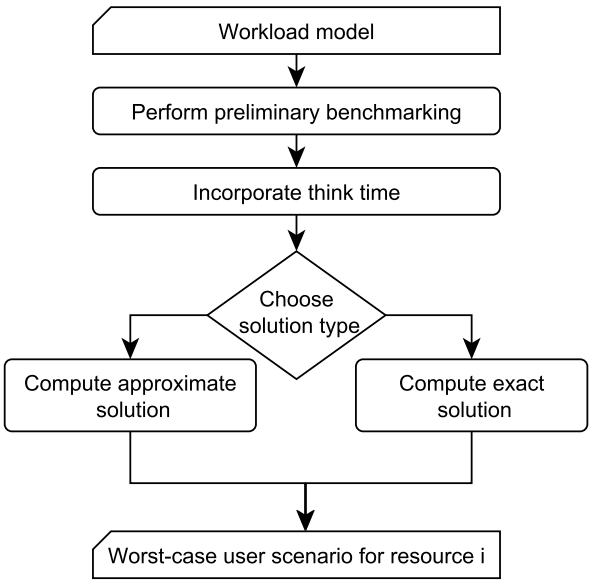

Figure 2: Approach for identifying the worst-case user scenario

the resource utilization of each action via code instrumentation; however, other means of collecting the resource utilization can be used in case one does not have access to the source code of the SUT. During the test session, each action in the model will be executed several times. For each action, we calculate the average utilization of the selected resource type for all its executions.

For instance, we have executed the workload model in Figure 1 for 60 seconds against the implementation discussed in the previous sections. Table 1 summarizes the resulting average CPU and memory utilization values for each action in the model and the number of times each action has been executed during the test session.

5.2. Incorporating the think time

Conceptually, the DTMC formalism imposes that a transition from one state to another is done in one unit of time. In our extended version of DTMC, each transition can have a different think time value. In order to incorporate

Table 1: CPU and Memory utilization with one concurrent user for Figure 1

Action	**CPU (sec)**	**Memory (MB)**	**Repetitions**
get_bids(id)	0.083	0.198	2324
search(string)	0.088	0.200	614
browse()	0.178	0.201	2185
get_auction(id)	0.072	0.199	2416
bid(id,price,username,password)	0.578	0.202	1176

different think time values into DTMC according to its definition, we convert the workload model into an intermediate format, by unfolding the think time of a transition over several *pseudo-states* (i.e., nodes with zero resource utilization). In other words, we insert pseudo-states between two states, according to the specified think time on the transition, each pseudo-state corresponding to one time unit. We emphasize that adding pseudo-states in the DTMC will increase the length of a path between any two nodes proportionally with the think time value. However, it will not introduce additional loops or circuits in the model.

For example, we obtain the workload model in Figure 3 after incorporating the think time in the original workload model shown in Figure 1. The dotted circles in Figure 3 represent pseudo-states. For distinguishing between pseudo-states in future examples, we have labeled each of them with a number. Nevertheless, we would like to point out that the two models are semantically equivalent with respect to our performance testing process.

5.3. Method 1: Computing the worst path using graph-search algorithms

As a first alternative method, we suggest the use of graph-search algorithms for computing the worst path, as described in Algorithm 1. Three input parameters are given: the workload model in an intermediate format as a directed graph (G), benchmarked resource utilization of each node (U^r) for a given resource type, and the initial node in the graph ($INode$). The output will provide the worst path in the graph with respect to utilization of the selected resource type. The algorithm is deterministic and always returns the exact solution.

Algorithm 1 Graph-search based approach

1: **procedure** WorstPath($G, U^r, INode$)
2: $all\_circuits \leftarrow$ FindAllElementaryCircuits(G) ▷ Use Johnson's algorithm[20] to get a set of all elementary circuits
3: $worst\_circuit \leftarrow SelectMax(\{(c, CRU(U^r, c)) \mid c \in all\_circuits\})$ ▷ Select the *circuit* with the highest resource utilization
4: **if** $INode \notin worst\_circuit$ **then**
5: $short\_paths \leftarrow$ ShortestPathsFrom($G, INode$) ▷ Get all the shortest paths from the initial node to all the other nodes using Dijkstra's algorithm[21]
6: $short\_paths\_to\_cir \leftarrow \{p \mid p \in short\_paths \wedge p \cap worst\_circuit \neq \emptyset\}$
7: $min\_short\_path \leftarrow SelectMinLen(short\_paths\_to\_cir)$ ▷ Select the shortest path with the *minimal length*
8: $worst\_path \leftarrow$ MergePaths($min\_short\_path, worst\_circuit$)
9: **else**
10: $worst\_path \leftarrow worst\_circuit$
11: **end if**
12: **end procedure**
13: **function** CRU(RU, Path) ▷ Calculate resource utilization per node of the given *Path*
14: **return** Sum($\{RU[n] \mid n \in Path\}$)$\div \mid Path \mid$
15: **end function**
16: **function** MergePaths(path, circuit) ▷ Merge the given path with circuit
17: $node\_joint \leftarrow path \cap circuit$
18: $Q \leftarrow$ Queue($circuit$)
19: $top\_node \leftarrow$ Q.dequeue()
20: **while** $top\_node \neq node\_joint$ **do**
21: Q.Enqueue(top_node)
22: $top\_node \leftarrow$ Q.dequeue()
23: **end while**
24: **return** $path \cup Q$
25: **end function**

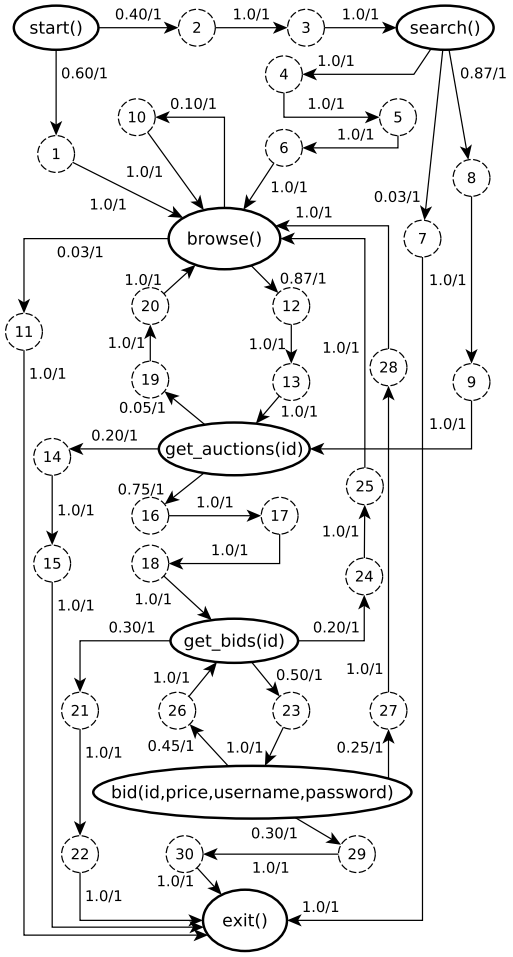

Figure 3: Incorporating think time into the workload model (shown in Figure 1)

The algorithm has three steps. In the first step (line *2*), we find all the elementary circuits in the model by running Johnson's algorithm [20] on the given workload model. An *elementary circuit* is a path in which no node, except the first and last, appears twice. To the best of our knowledge, Johnson's algorithm is the fastest algorithm to find the elementary circuits in a directed graph.

In the second step (line *3*), we calculate the average resource utilization of

each elementary circuit and we select the circuit which has the highest resource utilization as the *worst circuit*. In the third step (lines *4* to *8*), if the initial node is present in the worst circuit, then the circuit corresponds to the worst path in the workload model and we return it as a solution. If the initial node is not present in the worst circuit, we identify the shortest path from the initial node to any node in the worst circuit. To this extent, we use Dijkstra's [21] algorithm to find all the shortest paths (lines *5–6*) from the initial node to each node in the worst circuit. Then, we select the path with the minimum length among the identified the shortest paths (line *7*). Finally, we merge (line *8*) the selected shortest path with the worst circuit to get the worst path in the model. The reason for combining the shortest path with the worst circuit is that we want the virtual users to arrive at the worst circuit as quickly as possible during load generation. As a result, the virtual users will spend the majority of their simulation time in the worst circuit and generate more workload against the SUT.

When applied to the workload model in Figure 3 with respect to CPU utilization, the algorithm discovers *five* elementary circuits and selects the following circuit as the worst circuit in the model:

$$get\_bids(id) \rightarrow P_{23} \rightarrow bid(id, price, username, password) \rightarrow P_{26} \rightarrow get\_bids(id)$$

where P_{23} and P_{26} represent the pseudo-states in the workload model with label *23* and *26*, receptively. Since the worst circuit does not contain the initial node, the algorithm selects the following path as the shortest path from the initial node to the worst circuit:

$$start() \rightarrow P_1 \rightarrow browse() \rightarrow P_{12} \rightarrow P_{13} \rightarrow get\_auction(id) \rightarrow P_{16} \rightarrow P_{17} \rightarrow P_{18} \rightarrow get\_bids(id)$$

By connecting the shortest path with the worst circuit and after folding back the pseudo-states, we attain the following path as the worst path in the workload

model in Figure 1 with respect to CPU utilization:

$$start() \rightarrow browse() \rightarrow get\_auction(id) \rightarrow get\_bids(id)$$
$$\rightarrow bid(id, price, username, password) \rightarrow get\_bids(id) \quad (1)$$

Time Complexity Analysis of the Exact Method. Johnson's algorithm to find all the circuits in a directed graph has $O((N + E)(C + 1))$ [20] time complexity, where N, E and C represent the number of nodes, edges, and circuits in a given graph, respectively. The CRU function for selecting the worst circuit (line *3*) has $O(N)$ time complexity, which will be executed C times to calculate the resource utilization. Then, we select a circuit which has the highest resource utilization among all the circuits with the time complexity of $O(N)$. Therefore, the step at line *3* has $O(CN)$ time complexity. In case the worst circuit does not contain the initial node, we need to use the Dijkstra's algorithm [21] (at the line number *5*) with $O(E + NlogN)$ time complexity to calculate all the shortest paths from the initial node to all the other nodes in the model.

A summary of the time complexity of each step of the algorithm is shown in Table 2. Since the number of edges in a directed graph is upper bounded by $|E| = O(|N|^2)$, then we can conclude that the time complexity of the entire algorithm is dominated by the time complexity of the step at line 2, i.e., $O(N^2C)$. The time complexity is largely driven by the number of elementary circuits C in the model, which can grow faster than the exponential 2^N with respect to N [20].

As one can notice, even though this method produces the exact solution, it does not scale well for graphs with a large number of elementary circuits. Therefore, in the following section, we propose a heuristic method to compute an approximate solution, but with better scalability.

5.4. Method 2: Computing the near-worst path using genetic algorithms

As an alternative to the graph-search method, we propose the use of Genetic algorithms (GA) [22] to infer an approximate solution, that is a *near-worst path*

Table 2: Computational time complexity of Procedure *WorstPath* in Algorithm 1

Line number	**Time Complexity**
2	$O((N+E)(C+1))$
3	$O(CN)$
5	$O(E+NlogN)$
6	$O(N^3)$
7	$O(N^2)$
8	$O(N)$

in the DTMC with respect to resource utilization. GA is an optimization technique that is inspired by the natural evolution of species. The basic idea behind GA is to imitate the evolution of potential solutions for a given optimization problem.

When using GA, a *population* of *individuals* is maintained. In our approach, the individuals of the population are represented by workload models. Each workload model in a population is encoded into a *chromosome*. Each chromosome is a collection of *genes*. In our case, a gene encodes the probability distribution of the outgoing edges of a given state in the workload model. For instance, the model depicted in Figure 3 will be encoded into the following chromosome (for brevity, we only show the probability distribution of the P_1 pseudo-state whereas the other pseudo-states have been replaced with the "..." symbol):

$$\langle \overbrace{\langle 0.60, 0.40\rangle}^{start}, \overbrace{\langle 1.0\rangle}^{P_1}, \overbrace{\langle 0.87, 0.10, 0.03\rangle}^{browse}, \ldots, \overbrace{\langle 0.87, 0.10, 0.03\rangle}^{search}, \ldots,$$
$$\overbrace{\langle 0.20, 0.75, 0.05\rangle}^{get\_auctions}, \ldots, \overbrace{\langle 0.30, 0.50, 0.20\rangle}^{get\_bids}, \ldots, \overbrace{\langle 0.30, 0.25, 0.45\rangle}^{bid} \overbrace{\langle\rangle}^{exit} \rangle$$

where, for instance, genes $\langle 0.60, 0.40\rangle$ and $\langle 0.87, 0.10, 0.03\rangle$ describe the probability distributions of the *start()* and, respectively, *browse()* state.

An evolution process allows creating new generations of the population. The individuals of the first generation of the population are generated from the initial workload model by randomly changing the probability distributions of each state under the constraint that the sum of the probabilities in a gene must be equal to 1. From each generation, we select those individuals (workload models) based on a *fitness function* which ranks the models by the amount of time they spend in the states with high resource utilization. The next generation is obtained by applying genetic operators to the selected individuals to create new individuals. We repeat this step for a fixed number of generations. In the end, we select the individual with the highest fitness value in all populations as the *proposed solution*. The solution will provide an approximate solution to finding the worst-case user scenario, which we denote as the *near-worst path*.

The method is based on the following steps, as illustrated in Algorithm 2.

Algorithm 2 Pseudocode of genetic algorithm

```
1: procedure GA(I, P, N, C_p, M_p, M_r, U^r)
2:     Pop ← CreatePopulation(P, N) ▷ Randomly generate initial population of size P where the length of each chromosome is N
3:     for all Chromosome c ∈ Pop do
4:         Fitness(c, U^r)                          ▷ Calculate fitness of a chromosome
5:     end for
6:     for i ← 1 to I do                ▷ Evolve the initial population for I generations
7:         BinaryTournament(Pop)        ▷ Select chromosomes for the next generation
8:         TwoPointCrossover(Pop, C_p)  ▷ Use to two-point crossover operator based on C_p probability
9:         Mutate(Pop, M_p, M_r)        ▷ Mutate the individuals based on M_p probability
10:        for all Chromosome c ∈ Pop do
11:            Fitness(c)
12:        end for
13:    end for
14: end procedure
```

Evaluation of Fitness. A *fitness value* is computed (line *4*) for each individual in the population. This value correlates the expected level of resource utilization that a given individual (i.e., DTMC model) will create on the SUT. The fitness of an individual is computed in two steps. First, we create a transition matrix from the probability values represented in a given chromosome, then the fitness of the individual is calculated by the benchmarked resource utilization values for the selected resource type.

The transition matrix is then used to compute the *stationary distribution* (SD) of the given workload model. The stationary distribution of a DTMC with a transition matrix P is some probability vector π, such that,

$$\lim_{n\to\infty} vP^n = \pi$$

where v is any probability vector and n represents the power of matrix P. This implies that, in the long run, no matter what the starting state was, the probability of the Markov Chain model to be in state i at any given moment is approximately $\pi_i \in \pi$ for all i. In summary, computing the SD allows us to deduce which states in the model will be visited more frequently than the others based on their probability distributions.

We have used an iterative method, called the *power method*, to calculate the stationary distribution of a chromosome. *Iterative methods* are mostly used to compute the stationary distribution of large Markov Chains [23]. Additionally, these methods are less CPU and memory intensive [24, 23, 25]. The main disadvantage of using iterative methods is that they provide approximate solutions and it is not certain how many iterations are required to converge to the exact solutions [23].

For instance, the stationary distribution vector π of the model in Figure 3 is calculated to be as follows (for brevity, we only show the stationary distribution of the P_1 pseudo-state whereas the other pseudo-states have been omitted):

$$\pi = \langle \overbrace{0.135}^{start}, \overbrace{0.195}^{P_1}, \overbrace{0.195}^{browse}, \ldots, \overbrace{0.054}^{search}, \ldots, \overbrace{0.178}^{get\_auctions}, \ldots, \overbrace{0.202}^{get\_bids}, \ldots, \overbrace{0.097}^{bid}, \ldots, \overbrace{0.135}^{exit} \rangle$$

One should note, that the inclusion in the workload model of the pseudo-states corresponding to think time values will influence the probability distributions of the other states in the model because, by inserting the pseudo-states, we increase the length of the path between any two states.

Secondly, we combine the SD with the benchmark results of resource utilization (discussed in Section 5.1) to calculate the fitness of a model. Let S be a set of all the states in the model M and let vector U^r be the CPU utilization of the action of each state $s \in S$ with respect to resource r. Then we can define the fitness of model M with respect to resource r as follows:

$$f_M^r = \sum_{s \in S} \pi_s U_s^r$$

Considering as example the benchmarked utilization values for CPU included in Table 1 and the intermediate model shown in Figure 3, the resource utilization vector for CPU will be:

$$U^{CPU} = \langle 0.0, 0.0, 0.083, \ldots, 0.088, \ldots, \quad 0.178, \ldots, 0.072, \ldots, 0.578, \ldots, 0.0 \rangle$$

Then, the fitness of the model would be the by-product of the two vectors:

$$\begin{aligned} f_M^{CPU} &= 0.135 \times 0.0 + 0.195 \times 0.0 + 0.195 \times 0.083 + \ldots + 0.054 \times 0.088 + \ldots \\ &\quad + 0.178 \times 0.178 + \ldots + 0.202 \times 0.072 + \ldots + 0.097 \times 0.578 \\ &\quad + \ldots + 0.135 \times 0.0 = 0.123 \end{aligned}$$

We remind the reader that a pseudo-state does not represent an interaction between a user and the SUT. Since the resource utilization of a pseudo-state is always zero, it does not impact on the fitness value of an individual.

Applying the Selection operator. We use the *binary tournament* selection method in which two individuals are randomly picked and, between them, the individual with the larger fitness value is selected as the parent. Two chosen parents will generate two offspring.

Applying the Crossover operator. We use the *two-point crossover* operator to create offspring. The chromosomes of two parents are cut at two random points and the genes between the cut points are swapped to generate two offspring. The usage of the crossover operator is moderated by the *crossover operator probability* (C_p), which specifies the probability of applying the operator to the parents. For instance, by applying the two-point crossover operator to the parent chromosomes *V1* and *V2* in Figure 4, we obtain two chromosomes of their offspring, *W1* and *W2*.

Applying the Mutation Operator. A *mutation operator* is applied to the newly generated offspring in order to inject gene diversity in the population. The application of the operator is controlled by two parameters: *mutation operator rate* (M_p), which defines the probability of applying the operator to a given individual and *mutation rate* (M_r), which states the probability of modifying the probability distribution of a state in a selected individual. The newly generated offspring supersede the parents in the population.

In Figure 4, the mutation operator is applied to the newborn offspring *W1* and *W2* to obtain the final set of children *W1'* and *W2'*.

Calculating the near-worst path. Once the individuals of all generations have been created the one with the highest fitness value in all generations is selected as a solution. Its chromosome will provide the probability distributions of the workload model.

In order to find the near-worst case path in the model, we walk through the model starting from the initial state and, in each state, we select the outgoing edge with the highest probability. We stop when we arrive at a state which has been already visited.

For example, Figure 5 depicts a workload model in compact form (i.e., without having the think times expanded into pseudo-states) which is the result of applying our approximate method with respect to CPU utilization. In order to extract the most probable path, we start our walk from initial state in the model, *start()*. Then, we select an outgoing edge with the highest probability (i.e.,

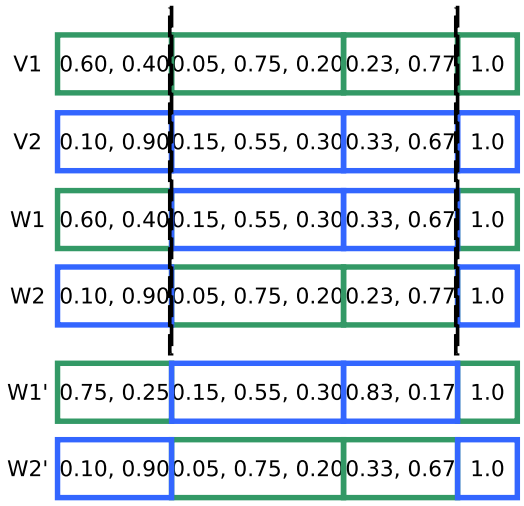

Figure 4: Generic example of applying the two-point crossover and mutation operators

0.95) and move to *browse()*. After visiting *get_auctions(id)*, *get_bids(id)*, and *bid(id,price,username,password)*, we select the edge with the highest probability 0.92 in state *bid(id,price,username,password)* and we walk back to *get_bids(id)* state, which, being visited already, constitutes the final state of our walk. Hence, the most probable path will be:

$$start() \rightarrow browse() \rightarrow get\_auction(id) \rightarrow get\_bids(id)$$
$$\rightarrow bid(id, price, username, password) \rightarrow get\_bids(id)$$

As one may notice, using the approximate method has returned exactly the same worst path as the exact method for this particular example. However, the approximate method does not always result in the exact solution.

Time Complexity Analysis of Approximate Method. Generally, the overall time complexity of a genetic algorithm is dictated by the fitness function [26]. Therefore, we only focus on the complexity of the fitness function when analyzing the complexity of our approach.

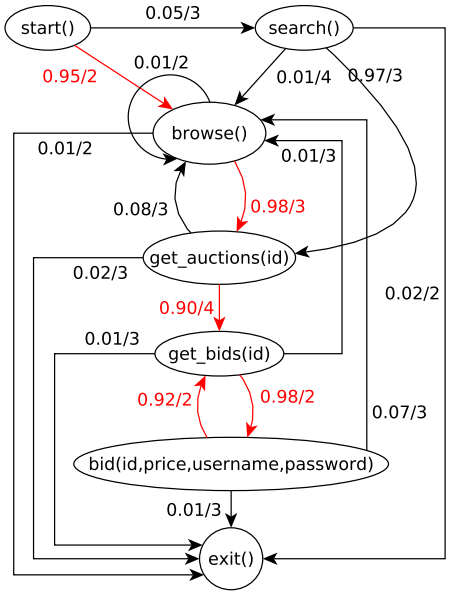

Figure 5: User model optimized for CPU utilization

We calculate the fitness of a workload model in two steps. Firstly, we transform the model into a probability transition matrix of size $N * N$, in N number of steps. Then, the *power method* is applied over the transition matrix to iteratively compute the stationary distribution vector [23]. Each iteration of the power method performs one matrix multiplication operation. The time complexity of a matrix multiplication operation is $O(N^3)$ for dense matrices and $O(EN)$ for sparse ones [27], where E is the number of edges in the corresponding graph. A matrix is considered sparse if $|E| = O(|N|)$ [28], which lets us conclude that the matrix multiplication for sparse matrices has a time complexity of $O(N^2)$.

Two conclusions can be drawn from the reasoning above: (1) the time complexity for calculating the approximate solution does not depend on the number of elementary circuits in the graph which can grow faster than the exponential 2^N with respect to N [20]; (2) the time complexity of the algorithm is bounded by $O(N^3)$ for dense graphs and by $O(N^2)$ for sparse ones. Both these conclusions show a clear performance improvement over the algorithm used for the

exact solution.

5.5. Tool support

All the steps of our approach presented in Figure 2 have been fully automated using existing open source libraries. The DEAP [29] evolutionary computation framework has been used to deploy the genetic algorithm. For benchmarking and for the evaluation of the approach we have used our MBPeT tool.

6. Experimental Validation and Evaluation

In this section, we perform several experiments to validate and evaluate our approach in order to answer the research questions posed in the introduction of this article. We will use the same *auction web application* described in Section 2 as a SUT. The web application has a *RESTful* [30] interface, based on the HTTP protocol. The web application is implemented using Python [31] and the Django [32] framework.

Validation. In the first experiment, we validate that the solutions identified by our approach are able to create the highest resource utilization on the SUT by comparing the resource utilization triggered by the worst path against the initial workload model and several random variants of the initial workload model. For this purpose, we have applied both methods on the initial workload model in Figure 1 and, as discussed in the previous section, both methods identified the same worst path. A new workload model was created to contain only the worst path (i.e., the transitions marked in red and the corresponding states in Figure 5), having all transition probabilities set to *1*, known as the *worst path model*. Additionally, we have generated nine random variants of the initial workload model by randomly modifying the probability distribution of the model.

We have used the MBPeT tool for load generation. The tool and the SUT ran on different machines. Each machine featured an Intel® Core™ i7-3770K CPU, 16 GB of memory, 7200 rpm hard drive, and Ubuntu 14.04 operating

system. In order to reduce the network latency, the machines were connected via a 1Gb Ethernet connection in an isolated environment. The SUT did not support automatic scalability and had a fixed number of resources available.

We have used the worst path model, the initial workload model in Figure 1, and nine random variants of the initial workload model to run 11 separate load generation sessions. Each session was run for 300 seconds. The ramp function displayed as a black dotted-line in Figure 6 was used for each session. The ramp specified that the tool kept increasing the number of virtual users to achieve 100 concurrent virtual users within 50 seconds and, after that, the number of virtual users was kept constant until the end of the session.

Figure 6 illustrates the results of all load generation sessions. The blue and green line show the resulting CPU utilization, respectively, for the worst and initial workload model. Additionally, the gray lines in the figure show the CPU utilization of the random variants of the initial workload model. The average CPU utilization was approximately 16% for the initial workload model and its random variants, and 54% the worst path model, which is almost three times higher.

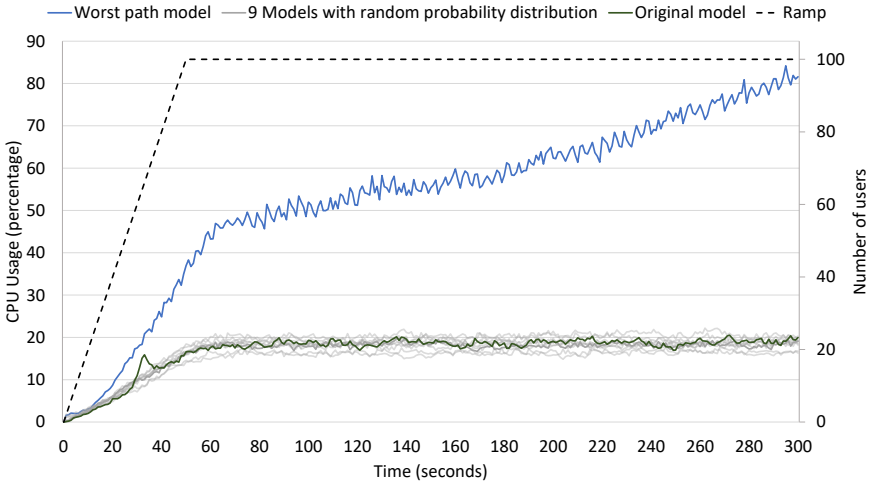

Figure 6: CPU utilization created by the models

The results of this experiment provide an empirical answer to *RQ1* showing that the solutions identified by both methods can identify the worst-case user scenario with respect to the utilization of a given resource type.

Evaluation. In the second experiment, we have evaluated the scalability of each method. For this purpose, we have randomly generated several sparse workload models with different number of nodes, as listed in Table 3. Each transition in the generated models is labeled with a random think time value and a dummy action, which triggers a certain amount of hypothetical resource utilization. For each generated model we calculated the number of elementary circuits, which, as seen in Table 3, increases dramatically with the size of the model. Each model was used to compute the worst path using both methods on a machine that features an Intel® Core™ i7-3770K CPU, 16 GB of memory and Ubuntu 14.04 Operating System. We have used the following parameters for every GA run:

- Population size (P) = 200
- Number of generations (I) = 50
- Crossover operator probability (C_p) = 0.3
- Mutation operator probability (M_p) = 0.5
- Mutation rate (M_r) = 0.01

Figure 7 provides an overview of the scalability of both methods with respect to their execution time. As it can be observed, the execution time of the exact method increases sharply for models with more than 20 nodes. For instance, the execution time for the model with 21 nodes was around 247 seconds, whereas, it took 31 minutes for the model with 22 nodes before crashing due to the insufficient memory space of the system. At the time of the crash 68 368 711 elementary circuits were counted. Since we could not compute the exact solution for the model with 22 nodes, we have neither listed the model in Table 3 nor used it in any further experiments.

In contrast, the approximate method performs better than the exact one,

Table 3: Generated Sparse Models

Nodes	Edges	Elementary Circuits
10	23	25
12	42	270
14	50	2 030
16	73	53 211
18	85	189 776
19	95	907 861
20	103	6 141 014
21	111	12 764 464

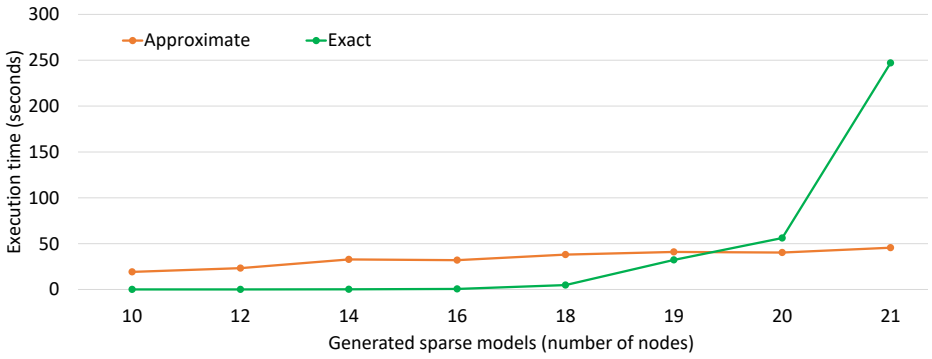

Figure 7: Execution times of the exact vs approximate method

being agnostic to the number of elementary circuits. The average execution time of the approximate approach was around 40 seconds for all workload models. In addition, one can further decrease the execution times of the approximate method by reducing the total number of generations or the population size, at the expense of less accurate results.

In the third experiment, we have evaluated the accuracy of the approximate method with respect to the exact method. For this purpose, we have measured the quality of the solutions derived from our approximate method and the con-

vergence rate of GA. We define the *quality of a solution* as the *ratio* between the fitness of an approximate solution estimated by the approximate method and the fitness of the exact solution computed using the exact method. In short, we will refer to it as the *quality ratio* of an approximate solution.

In the first step of the experiment, we have computed the exact solution for each model listed in Table 3 using the exact method. In the second step, we have computed the corresponding approximate solutions for the same set of models, using the same parameters for the GA as in the previous experiment. In order to reduce the random bias caused by the GA, we have applied the approximate method 10 times on each model listed in Table 3 while monitoring the convergence rate of each GA run. We define the *convergence rate of a GA run* as the rate of change in the average quality ratio of a population over all generations.

Figure 8 displays the convergence rate of 10 GA runs for every model listed in Table 3. As one can observe, the approximate approach converges quickly, in some cases, to the exact solution (i.e., with a quality ratio of 1). For instance, all GA runs for the models with 18 and 19 nodes converge to the exact solutions just after 20 generations.

Figure 9 shows that 59 out of 80 GA runs converged to the exact solutions (i.e., with a quality ratio of 1), in other words our approximate method returned the exact solution 73% of the cases.

The results of the second and third experiment provide an answer to our RQ2 that the exact method exhibits poor scalability for those models which contain a large number of elementary circuits, but produce the exact solution. However, the approximate method can be used with the larger models at the expense of producing an approximate solution.

7. Conclusion

We have proposed an approach for performance testing of web applications in which we identify the worst path (i.e., a sequence of user interactions) in a

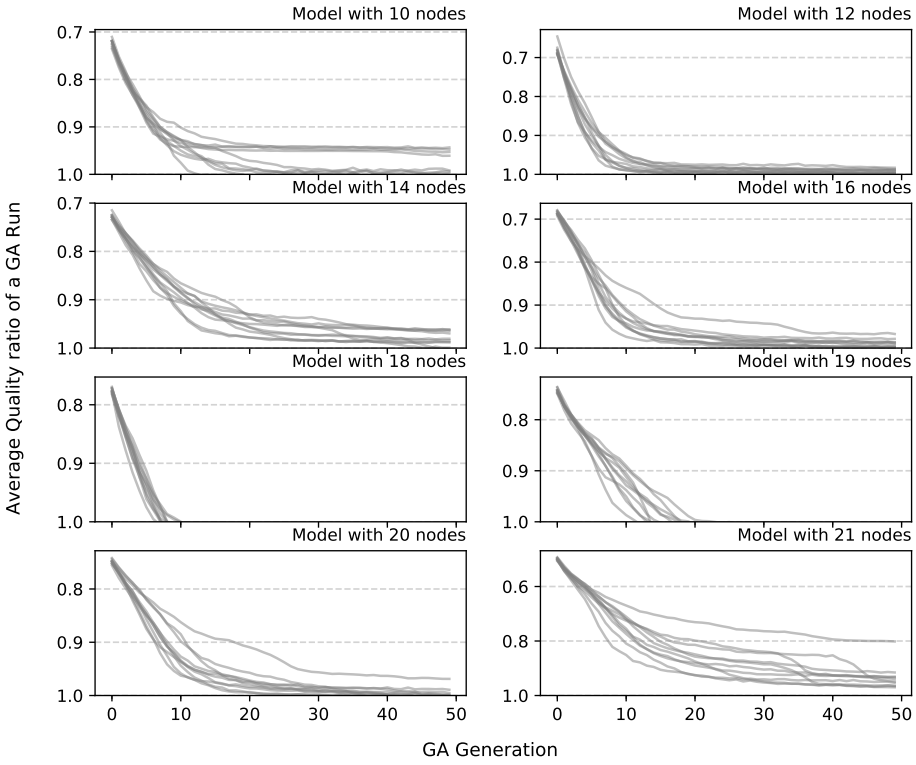

Figure 8: Convergence rate of 10 GA runs per model

given workload model which will cause the highest utilization of a given resource on the SUT. In order to answer *RQ1*, we have proposed an exact and an approximate method for detecting the worst path in the workload model. Then, in order to answer *RQ2*, we have analyzed both analytically and empirically the performance of the two methods.

Moreover, we have noticed that in the case of models with a large number of elementary circuits, the approximate method performs better. The reason is that the time complexity of the approximate method does not depend on the number of elementary circuits in the model as in the case to the exact method.

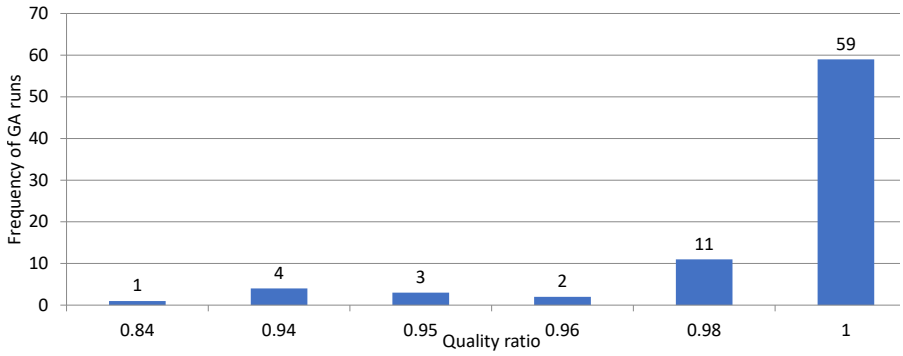

Figure 9: Frequency distribution of all GA runs based on the quality ratio

On the other hand, in the case of sparse models, the exact method will execute faster than the approximate method and provide the exact solution.

As a final remark, the two proposed methods have both advantages and drawbacks. Thus, the most appropriate one should be chosen depending on the configuration and complexity of the workload model under consideration.

Acknowledgments. This work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737494. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland, Czech Republic.

References

[1] Gomez, Why web performance matters: Is your site driving customers away?, retrieved: Decemeber 2017 (2010).
URL `http://www.mcrinc.com/Documents/Newsletters/201110_why_web_performance_matters.pdf`

[2] B. Subraya, S. Subrahmanya, Object driven performance testing of web applications, in: Quality Software, 2000. Proceedings. First Asia-Pacific Conference on, IEEE, 2000, pp. 17–26. `doi:10.1109/APAQ.2000.883774`.

[3] F. Abbors, T. Ahmad, D. Truscan, I. Porres, MBPeT A Model-Based Performance Testing Tool, in: 4th International Conference on Advances in System Testing and Validation Lifecycle, IARIA, 2012, pp. 1–8.

[4] F. Abbors, T. Ahmad, D. Truscan, I. Porres, Model-based performance testing of web services using probabilistic timed automata, in: Proceedings of the 2013 10th International Conference on Web Information Systems and Technologies, 2013.

[5] C. M. Grinstead, J. L. Snell, Introduction to probability, American Mathematical Soc., 2012.

[6] F. Abbors, D. Truscan, A. Tanwir, An automated approach for creating workload models from server log data, in: H. Andreas, L. Therese, M. Leszek, M. Stephen (Eds.), Proceedings of the 9th International Conference on Software Engineering and Applications, Scitepress, 2014, pp. 14–25. doi:10.5220/0005002200140025.

[7] A. van Hoorn, C. Vögele, E. Schulz, W. Hasselbring, H. Krcmar, Automatic extraction of probabilistic workload specifications for load testing session-based application systems, in: Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '14, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 2014, pp. 139–146. doi:10.4108/icst.valuetools.2014.258171.

[8] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, H. Krcmar, Wessbas: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems, Software & Systems Modeling (2016) 1–35doi:10.1007/s10270-016-0566-5.

[9] L. Slothouber, A model of web server performance, in: Proceedings of the 5th International World wide web Conference, 1996.

[10] J. Dilley, R. Friedrich, T. Jin, J. Rolia, Web server performance measurement and modeling techniques, Performance evaluation 33 (1) (1998) 5–26. doi:10.1016/S0166-5316(98)00008-X.

[11] M. S. Squillante, D. D. Yao, L. Zhang, Web traffic modeling and web server performance analysis, in: Decision and Control, 1999. Proceedings of the 38th IEEE Conference on, Vol. 5, IEEE, 1999, pp. 4432–4439. doi:10.1109/CDC.1999.833239.

[12] P. Reeser, R. Hariharan, An analytic model of web servers in distributed computing environments, Telecommunication Systems 21 (2) (2002) 283–299.

[13] Á. Bogárdi-Mészöly, T. Levendovszky, A novel algorithm for performance prediction of web-based software systems, Performance Evaluation 68 (1) (2011) 45–57. doi:10.1016/j.peva.2010.09.004.

[14] A. Gao, D. Yang, S. Tang, M. Zhang, Mining models of composite web services for performance analysis, Lecture notes in computer science 3882 (2006) 828. doi:10.1007/11733836_60.

[15] C. Stewart, K. Shen, Performance modeling and system management for multi-component online services, in: Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI'05, USENIX Association, Berkeley, CA, USA, 2005, pp. 71–84.
URL http://dl.acm.org/citation.cfm?id=1251203.1251209

[16] E. Hernández-Orallo, J. Vila-Carbó, Web server performance analysis using histogram workload models, Computer Networks 53 (15) (2009) 2727–2739. doi:10.1016/j.comnet.2009.06.005.

[17] J. Offutt, Quality attributes of web software applications, IEEE Softw. 19 (2) (2002) 25–32. doi:10.1109/52.991329.

[18] D. H. Bailey, A. Snavely, Performance modeling: Understanding the past and predicting the future, in: European Conference on Parallel Processing, Springer, 2005, pp. 185–195. doi:10.1007/11549468_23.

[19] T. Ahmad, D. Truscan, Automatic performance space exploration of web applications using genetic algorithms, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, ACM, New York, NY, USA, 2016, pp. 795–800. doi:10.1145/2851613.2851864.

[20] D. B. Johnson, Finding all the elementary circuits of a directed graph, SIAM Journal on Computing 4 (1) (1975) 77–84. doi:10.1137/0204007.

[21] M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (3) (1987) 596–615. doi:10.1145/28869.28874.

[22] M. Srinivas, L. M. Patnaik, Genetic algorithms: A survey, Computer 27 (6) (1994) 17–26. doi:10.1109/2.294849.

[23] W. J. Stewart, Introduction to the numerical solutions of Markov chains, Princeton Univ. Press, 1994.

[24] M. C. Seiler, F. A. Seiler, Numerical recipes in c: the art of scientific computing, Risk Analysis 9 (3) (1989) 415–416. doi:10.1111/j.1539-6924.1989.tb01007.x.

[25] D. Radev, V. Denchev, E. Rashkova, Steady-state solutions of markov chains, in: Proceedings of the 7th Balkan Conference on Operational Research, 2005.

[26] F. G. Lobo, D. E. Goldberg, M. Pelikan, Time complexity of genetic algorithms on exponentially scaled problems, in: Proceedings of the 2Nd Annual Conference on Genetic and Evolutionary Computation, GECCO'00, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, pp. 151–158.
URL http://dl.acm.org/citation.cfm?id=2933718.2933739

[27] R. Yuster, U. Zwick, Fast sparse matrix multiplication, ACM Transactions on Algorithms (TALG) 1 (1) (2005) 2–13. doi:10.1145/1077464.1077466.

[28] D. Jungnickel, Graphs, Networks and Algorithms, Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-32278-5.

[29] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy, Journal of Machine Learning Research 13 (2012) 2171–2175. doi:10.1145/2330784.2330799.

[30] L. Richardson, S. Ruby, Restful web services, 1st Edition, O'Reilly, 2007.

[31] Python, Python programming language, Online at http://www.python.org/, retrieved: January, 2014 (2012).
URL http://www.python.org/

[32] Django, A high-level python web framework, Online at https://www.djangoproject.com/, retrieved: January, 2014 (2012).
URL https://www.djangoproject.com/