Taylor & Francis
Taylor & Francis Group

Applied Artificial Intelligence
An International Journal

ISSN: 0883-9514 (Print) 1087-6545 (Online) Journal homepage: https://www.tandfonline.com/loi/uaai20

# Metaheuristic Algorithm for State-Based Software Testing

Ramzi A. Haraty, Nashat Mansour & Hratch Zeitunlian

**To cite this article:** Ramzi A. Haraty, Nashat Mansour & Hratch Zeitunlian (2018) Metaheuristic Algorithm for State-Based Software Testing, Applied Artificial Intelligence, 32:2, 197-213, DOI: 10.1080/08839514.2018.1451222

**To link to this article:** https://doi.org/10.1080/08839514.2018.1451222

Published online: 09 Apr 2018.

APPLIED ARTIFICIAL INTELLIGENCE
2018, VOL. 32, NO. 2, 197–213
https://doi.org/10.1080/08839514.2018.1451222

Taylor & Francis
Taylor & Francis Group



# Metaheuristic Algorithm for State-Based Software Testing

Ramzi A. Haraty, Nashat Mansour, and Hratch Zeitunlian

Department of Computer Science and Mathematics, Lebanese American University, Beirut, Lebanon

**ABSTRACT**

This article presents a metaheuristic algorithm for testing software, especially web applications, which can be modeled as a state transition diagram. We formulate the testing problem as an optimization problem and use a simulated annealing (SA) metaheuristic algorithm to generate test cases as sequences of events while keeping the test suite size reasonable. SA evolves a solution by minimizing an energy function that is based on testing objectives such as coverage, diversity, and continuity of events. The suggested method includes a "significance weight" assigned to events, which leads to important web pages and ensures coverage of relevant features by test cases. The experimental results demonstrate the effectiveness of simulated annealing and show that SA yields good results for testing web applications in comparison with other heuristics.

## Introduction

Web applications have evolved significantly in the recent decade. A new dimension of web technology, known as Web 2.0, is depicted where web applications are no longer static pages but lighter client applications. In Web 2.0, the web is approached as a platform, and software applications are built upon the web as opposed to being built upon the desktop (O'Reilly 2005). Web 2.0 applications are heavily built around several technologies such as AJAX, rich media content, widgets, and third-party applications that can be executed within webpages, Webparts, Portlets, and similar HTML units. Applications developed with AJAX technology provide the user with a rich dynamic interface that enables responsive interaction through light client software where the user is capable of controlling the content of the website through asynchronous requests and responses resulting in a new page that is updated dynamically through the Document Object Model (DOM).

The new web technology introduces additional challenges to the already hard task of web application testing. In addition to the searchability and accessibility, we have to test the dynamic user interface elements and states to find abnormalities and errors (Marchetto, Tonella, and Ricca 2008). For example, with Web 2.0, not

**CONTACT** Ramzi A. Haraty rharaty@lau.edu.lb Department of Computer Science and Mathematics Lebanese American University Beirut, Lebanon 1102-2801.

Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/UAAI.

every dynamically generated page has a unique uniform resource locator (URL). Furthermore, not every state change has a distinct uniform resource identifier (URI), and therefore, not all navigation paths to different states of web pages are available. Thus, existing web testing methods (Andrews, Offutt, and Alexander 2005; Di Lucca et al. 2002; Tarhini., Mansour, and Fouchal 2010) are not adequate to test Web 2.0 applications.

In this article, we propose an effective state-based testing method, which can be used for the Web 2.0 application. For example, this method can be based on deducing web page states and generating the equivalent state transition chart. Then, we use a metaheuristic approach based on simulated annealing (SA) to simultaneously generate a controlled number of test cases with maximum diversity and coverage. SA is a single-solution-based well-established metaheuristic that has been used for solving many real-world problems. It has exhibited faster processing than population-based metaheuristics (Mansour, Isahakian, and Ghalayini 2011). Although the method will be presented for and applied to examples of web applications, it is appropriate for software applications that can be modeled by "weighted" state graph.

In the next section, we review related works on testing web applications. In Section 3, we present the specificities pertaining to testing web applications. Section 4 describes how to build a state-based model for web applications. In Section 5, we present the simultaneous-operation simulated annealing (SO-SA) algorithm. Experimental results are discussed in Section 6, and in Section 7, we conclude the article.

## Related work

In an effort to reduce application testing costs and improve software quality, numerous works have been done on automating testing techniques (Ferguson and Korel 2006). One of the approaches used to automate test case generation is based on state machine model or even flow model (Nikolik 2006). State-based testing is ideal when dealing with sequences of events. In some cases, the sequences of events can be potentially infinite, which of course exceeds testing capabilities; thus, there is the need to come up with a design technique that allows handling sequences of random lengths. State-based testing model has proved to be a successful approach especially when dealing with graphical user interface (GUI) testing. However, the approach is considered resource intensive especially while generating the model due to the significant manual intervention needed. To improve the cost-effectiveness of the method and reduce the number of possibilities, state-based testing is extended to be formulated on a feedback strategy. When using state machines to model a Web 2.0 application, states represent the user interfaces and the state transitions represent the events triggering the transition. A test case is a sequence of events that correspond to a path in the finite state

machine (FSM). FSM representation of Web 2.0 applications, like all modern applications, has a scaling problem because of the large number of candidate states and transitional events. Several suggestions were proposed by researchers to handle the scalability issue based on path search algorithms. Several variants of FSMs have also been used for testing. The mutations are driven from the main aim to reduce the total number of states, and algorithms traverse these machine models to generate sequences of events as test cases. These techniques require an initial test suite to be created, either manually or automatically. Then, the test suite is to be executed and subsequently evaluated. The feedback resulting from the evaluation is used to permute the initial configuration to automatically enhance or generate new test cases. The evaluation of feedback strategy is formulated mainly around the optimization algorithm used to target a specific goal. The targeted goal can be one of many; however, usually it is code coverage, state coverage, or diversity to improve the overall performance of the test suite (Kolawa and Huizinga 2007; Mansour, Zeitunlian, and Tarhini 2013).

In contrast to the FSM, which can be used to generate test suites that guarantee complete fault coverage, or a complete test suite within the bounds to detect mutant FSMs within a predefined number of states, an extended finite state machine (EFSM) can often be viewed as a compressed notation of an FSM. It is possible to unfold it into a pure FSM by expanding the values of the parameters, assuming that all the domains are finite. However, this expansion should be carefully designed so as not to fall into the same trap of state explosion. Petrenko and Boroday (2004) call the state of unfolded EFSM as "configuration" and investigate the problem of constructing a configuration of sequences from an EFSM model, specifically when unfolded EFSM states result in generation of sequences that are different from sequences obtained from the initial configurations or at least they are not in the maximal subset. The authors generalize the problem into a search problem-generating configurations sets. They demonstrate how the problem can be tackled and EFSM reduced so that existing testing methods, which rely on FSM, can handle the configurations as input. They present a theoretical framework for determining configuration-confirming sequences based on EFSMs. Moreover, they elaborate on different derivation strategies. The authors argue that the proposed approach of confirming sequence generation can be used to improve any existing test derivation tool that typically uses a model checker mainly to derive executable preambles and postambles.

Memon and Pollack worked on artificial intelligence planning to manage the state-space explosion by eliminating the need for explicit states (Memon, Pollack, and Soffa 2001). In their work, the GUI description is manually created by a tester in the form of planning operators, which model the preconditions and postconditions of each GUI event. The planner automatically generates test

cases using pairs of initial and destination transitional states. The authors proved the efficiency of the system and suggested that it be integrated with all FSM-based modeling techniques.

Liu et al. (2000) proposed a formal technique that models web application components as objects and generates test cases based on data flow between these objects. Ricca and Tonella (2001) presented a test generation model based on the unified modeling language. These techniques extend traditional path-based test generation and use forms of model-based testing. They can be classified as "white-box" testing techniques since the testing models are generated from the web application code.

The major challenges for the techniques of testing web applications with dynamic features are how to model the application and what algorithm can be used in order to select the test cases from a huge range of possibilities. Not much research has been reported on testing web applications with dynamic features using state transition diagrams. Marchetto, Tonella, and Ricca 2008 proposed a state-based testing technique designed to address the new features of Web 2.0 applications. In this technique, the DOM manipulated by AJAX code is abstracted into a state model where callback executions triggered by asynchronous messages received from the web server are associated with state transitions. The test cases are generated from the state model based on the notion of semantically interacting events. Empirical evidence shows the effectiveness of this type of testing in finding faults. However, this technique generates a very large number of test cases that could limit the usefulness of the test suites. Another proposal by Marchetto et al. (Marchetto, Tonella, and Ricca 2009) addressed this problem. They proposed a search-based approach based on a hill-climbing algorithm to generate test sequences while keeping the test suite size reasonably small. In order to preserve a fault revealing power comparable to that of exhaustive test suite, they aimed to maximize the diversity of the test cases by introducing a measure of test case diversity instead of exhaustively generating all test cases up to a given length $K$ and by selecting the most diverse test cases, without any constraint on their length $K$.

The industry also proposed several functional testing tools for testing web application. Some tools rely on capture/replay facilities, which allow functional testing (Web Application Testing Tools). They record the interactions that users have with the graphical interface and repeat them during regression testing. However, they do not detect the failures in meeting the functional requirements. Other tools rely on discovering and systematically exploring website execution paths that can be followed by a user in a web application (Benedikt, Freire, and Godefroid). Further approaches to functional testing are based on user session data to produce test suites (Elbaum et al. 2005). Others are based on HttpUnits where the application is divided to HttpUnits and tested by mimicking web browser behavior (Fejes). HttpUnit can be used for unit testing and it is best suited for the implementation of functional tests and

acceptance tests; however, it is not practical for typical web layer components such as JSP pages, servlets, and other template components.

## Testing Web applications

Testing is an essential part of the software development cycle. It is used to detect errors and to ensure the quality of the software. Web applications differ from traditional software development where they follow the Agile software development model (Szalvay 2004), which has shorter development time. Because of the short development time, web applications usually lack necessary documents during the development and the user requirements often change, and testing and maintaining web applications becomes a more complex task compared to traditional software.

During the past decade, radical changes were introduced to the development of web applications and even the concept of the web. The web is approached as a platform where software applications are built upon, hence, the emergence of a new generation of web applications and web systems known as Web 2.0. Web 2.0 applications are based on highly dynamic web pages, build around AJAX technologies, which, through the asynchronous server calls, enable the users to interact and affect the business logic on the servers. AJAX technology created an umbrella under which the Web 2.0 applications are able to provide high level of user interaction and web page dynamics. Google Maps, Gmail, and Google Documents are a few examples of Web 2.0 applications.

The objective of our research is to develop a more effective state-based testing for a Web 2.0 application that will cover its dynamic features. This testing approach is based on a search-based algorithm rather than exact graph algorithms for traversing the events in the state-based graph model.

## State graph modeling

Extracting a state graph from a Web 2.0 application is not a direct and simple task. The main challenge is the absence of traditional navigational paths. This is because in Web 2.0, there is no unique URI assignment to a specific variant of the dynamic page, unlike traditional web applications where each web page state in the browser has an explicit URI assigned to it. Moreover, an entire Web 2.0 application can be created from a single web page where the user interface is determined dynamically through changes in the DOM initiated by user interaction through asynchronous server calls. Furthermore, the Web 2.0 application may contain third-party HTML units, user-shared data, widgets, and media content that are added to the application simultaneously during execution. To overcome the above-mentioned challenges, our testing mechanism will reconstruct the user interface states and generate static pages having navigation paths each with a unique URL. These static pages will be used to

conduct state-based testing (Ricca and Tonella 2001). To attain the static like pages, we need a tool that will execute client-side code and identify clickable elements which may change the state HTML/DOM within the browser. From these state changes, we will build our state graph that captures the states of the user interface, and the possible transitions between the states.

### Building the state graph

Our model reveals the user interface state changes in the Web 2.0 application. Thus, the model records all navigation paths/semantically interacting event of the DOM state changes. This is represented by a state graph for a Web 2.0 site A, which is a 4 tuple $<r,V,C,E>$ where:

(1) $r$ is the root node representing the initial state after $A$ has been fully loaded into the browser.
(2) $V$ is a set of vertices representing the states. Each $v \in V$ represents a run-time state in $A$.
(3) $C$ is a set of clickable elements that enable the transition from one state to another.
(4) $E$ is a set of edges between vertices. Each $(v_1,v_2) \in E$ represents a clickable $c \in C$ connecting two states, if and only if state $v_2$ is reached by executing $c$ in state $v_1$.

Events $e_1$ and $e_2$ are interacting semantically if there exists a state $S_0$ such that their execution in $S_0$ does not commute; i.e., the following conditions hold:

$$S_0 = >_{e1:e2} S_1$$
$$S_0 = >_{e2:e1} S_2$$
$$S_1 <> S_2$$

where $S_0$, $S_1$, and $S_2$ are any states in the state graph of the web application. The notion of a pair of semantically interacting events can be easily generalized to sequences. The event sequence $(e_1 \ldots e_n)$ is a sequence of a semantically interacting event if every pair of events in the sequence is a pair of semantically interacting events.

Figure 1 depicts the visualization of the state graph for a simple Web 2.0 application responsible for managing online photo albums. Its main functionalities are creating an album, deleting an album, selecting an album, editing an album, saving an album, adding photo, deleting photo, and displaying album. Moreover, the figure shows the three main states of the application: *S1*, starting state; *S2*, where at least one photo is selected; and *S3*, an album is selected. It illustrates how the three different states can be reached.

Two issues are to be considered while building the state graph. First, we need to detect the event-driven elements; next, we need to identify the state changes. The state graph is created incrementally; initially, the state graph contains only the root state. Additional states are appended to the graph as event-driven elements are traced/invoked in the application and state changes are analyzed. In the following subsections, we detail how event-driven elements are detected and state changes are identified.

In order to detect even-driven elements, we suggest a candidate list of elements that responds to events (clickable events). For example, <*div*>, <*input*>, <*a*>, and others may respond to events like (*Click, Doubleclick*, and *Mouseover*). Once an HTML page is loaded, we access the HTML elements through the DOM and detect the event-driven elements by checking whether each element in the DOM belongs to the suggested candidate list of clickable events.

Once a candidate element is detected, we execute the event attached to that element. In order to determine whether the execution of the event results in state change, we compare the DOM-tree version after executing the event and the DOM-tree version just before executing that event. If the execution of the event results in a state change, we check whether the resulting state is already covered in the graph. If the state was not previously covered, the new state is added to the graph, and an edge representing the executed event will connect the two states. If the state is already covered, an edge will be added between the states. Figure 2 shows the DOM-tree before an event is executed. This state is characterized by having the entire HTML element set to null or empty. Figure 3 shows the state of the DOM after "Select Album" event is executed. Note the change in the element <*span id = "lblAlbumName*"> before and after the event execution; such a state is identified as a new state. The obtained state, being a new state, will be added to the FSM and an edge marking the event select album will be added between the states. Next, the clickable element <*input type = "submit" name = "btnShowAlbum"*> is detected and the "Show Album" event is executed. The execution of this event forces dynamic changes to DOM and a transition to a new state. The new DOM state representation is depicted in Figure 4. The figure shows that at least one photo —an image element with nonempty image source and text element containing the description of the photos in addition to the name of the album. Comparing this state with the previously obtained states, it is marked as new and added to the FSM. Similarly, all clickable elements will be detected and executed, and the FSM generated covering all functionalities of the Web 2.0 application is included.

## Simulated annealing algorithm

The SA algorithm simulates the natural phenomenon by a search (perturbations) process in the solution space optimizing some cost function (energy) (Kirkpatrick, Gelatt, and Vecchi 1983). It starts with some initial solution at a high (artificial) temperature and then reduces the temperature gradually to a

freezing point. In the following subsections, we describe how we generate test sequences of semantically interacting events using the SA algorithm; an outline of the SA algorithm is given in Figure 5. In our work, we choose SA, in contrast with hill climbing (Marchetto, Tonella, and Ricca 2008), in order to generate test sequences because it allows uphill moves, which will force the solution to jump out of a local minimum in a controlled way. Interested readers are directed to Zeitunlian (2012) for more information regarding the details of the algorithm.

### Solution representation

Our proposed solution is represented as a configuration $C$, which is implemented as an array of variable-length test cases. Each test case is represented by a sequence of a maximum of $K$ events derived from the state graph. The length of the array is $K \times N$, where $N$ is the maximum number of test cases required in the solution. To allow the variable length of test cases, we introduce a random number of fake edges into our set of valid events. These fake edges, called "No Edge," will play the role of space holder in the array.

### Energy function

The energy function measures how good the current configuration is. We based the energy function on three major weighted factors. The weights represent the importance of each factor. The three factors are continuity, diversity, and coverage.

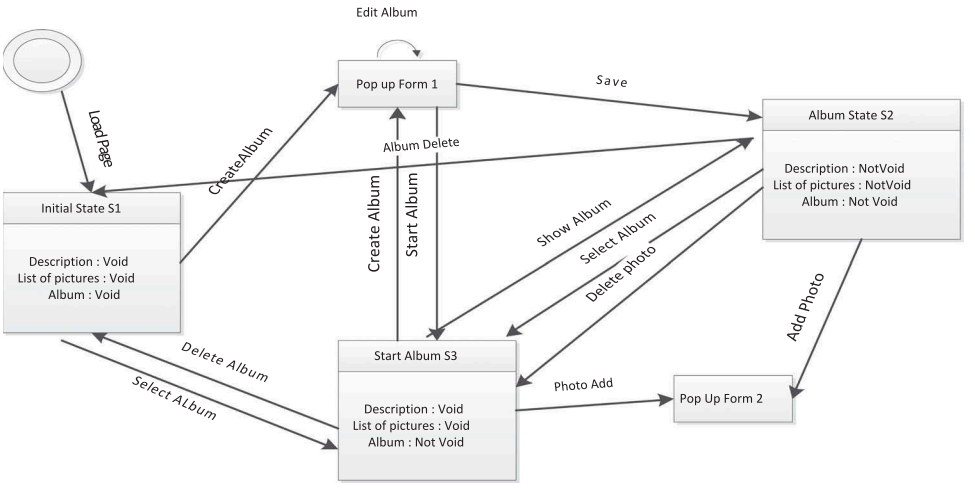


**Figure 1.** An example of a state graph model of a web application.

### Continuity

When testing event-based applications, it is very important to test a continuous set of events. In fact, test cases with longer continuous sequences of events have higher capability of revealing faults. In our SA strategy, we want to minimize/eliminate the discontinuity (*DC*) of events in a test case. We calculate discontinuity by checking the events in every test case and incrementing the value by one whenever discontinuous events are found.

### Diversity

Diversity is an important factor which guarantees that test cases will cover events from the entire scope of the web application, and not just concentrate on events from a certain part. Hence, we guarantee equally distributed events within the entire test suite. In this work, we will be minimizing the lack of diversity by calculating the average frequency of events in the entire test suite. Thus, given a test suite *S*, composed of a set of test cases based on semantically interacting sequences of events, its lack of diversity (*LDiv*) is computed as follows:

$$iv = \sqrt{\sum_{e \in Events} \left(F_e - F_{avg}\right)^2}$$

```
▼<div id="upAjaxContent">
  ▼<div>
     <input type="submit" name="btnSelect" value="Select" id="btnSelect">
     <input type="submit" name="btnDelete" value="Delete" id="btnDelete">
     <input type="submit" name="btnEdit0" value="Edit" id="btnEdit0">
     <input type="submit" name="btnShowAlbum" value="Show Album" id="btnShowAlbum">
    ▼<div>
      ▼<div>
         <span id="lblAlbumName"></span>
         <img id="Image1" src style="width:500px;">
         <textarea name="txtDescription" rows="2" cols="20" id="txtDescription" style="height:75px;width:
         499px;"></textarea>
       </div>
     </div>
   </div>
 </div>
```

**Figure 2.** Initial state—no album selected.

```
▼<div id="upAjaxContent">
  ▼<div>
     <input type="submit" name="btnSelect" value="Select" id="btnSelect">
     <input type="submit" name="btnDelete" value="Delete" id="btnDelete">
     <input type="submit" name="btnEdit0" value="Edit" id="btnEdit0">
     <input type="submit" name="btnShowAlbum" value="Show Album" id="btnShowAlbum">
    ▼<div>
      ▼<div>
         <span id="lblAlbumName">Lunch with friends</span>
         <img id="Image1" src style="width:500px;">
         <textarea name="txtDescription" rows="2" cols="20" id="txtDescription" style="height:75px;width:
         499px;"></textarea>
       </div>
     </div>
   </div>
 </div>
```

**Figure 3.** Start Album state—an album is selected.

```
▼<div id="upAjaxContent">
  ▼<div>
     <input type="submit" name="btnSelect" value="Select" id="btnSelect">
     <input type="submit" name="btnDelete" value="Delete" id="btnDelete">
     <input type="submit" name="btnEdit0" value="Edit" id="btnEdit0">
     <input type="submit" name="btnShowAlbum" value="Show Album" id="btnShowAlbum">
    ▼<div>
     ▼<div>
        <span id="lblAlbumName">Lunch with friends</span>
        <img id="Image1" src="http://localhost:59327/WebSite2/lunchtime%20011.jpg" style="width:500px;">
        <textarea name="txtDescription" rows="2" cols="20" id="txtDescription" style="height:75px;width:
        499px;">Hratch and George sitting at the balcony enjoying their lunch</textarea>
      </div>
     </div>
    </div>
  </div>
```

**Figure 4.** DOM of Album state—at least one picture selected.

where $e$ is an event that belongs to the set of events *Events*, $F_e$ is the execution frequency of event $e$, and $F_{avg}$ is the average frequency of event $e$ computed over the entire test suite.

### *Weighted coverage*

In Web 2.0 applications, end users and third parties can change the content of a web page dynamically by injecting HTML code or web widgets through their interaction with the site. Thus, some events would have higher importance than other events. Accordingly, we may control or even limit some functionality from being included in our testing plan by allowing a measure of importance of events that are part of the original web application, compared to injected events or functionality into the web application. The importance of events is represented by predefined weights assigned to every event. Again, we want to minimize the value of the unimportant events, and this value is calculated by checking if an event is covered in the test suite and multiplying it with its importance or weight. The weighted coverage is given by

$$WC = \sum (W_e * \ C_e), \text{where } e \ \varepsilon \text{ Events}$$

The energy function, $E$, is represented as:

$$\mathrm{E} = \alpha \times \frac{1}{WC} + \beta \times LDiv + \gamma \times DC$$

where α, $\beta$, and $\gamma$ are user-defined weights for weighted coverage, diversity, and discontinuity, respectively. Note that different values can be assigned to the weights in $E$. These weights are important for selecting test cases. They might be contradictory; that is, by increasing one of these weights, say $\alpha$, the solution will improve in minimizing one factor (discontinuity) while it might increase the other factors. These weights will allow flexibility in using our proposed algorithm to suit the user's particular choices or requirements for different instances of the problem.

```
initial configuration = sequence of events from the state graph;
determine initial temperature T(0);
determine freezing temperature Tf;
while (T(i) > Tf and not converged) do
  repeat
    generate_function();
  until several times(multiple of the number and size of required test cases)
  save_best_sofar();
  T(i) = θ * T(i);
endwhile

procedure generate_function()
perturb();
if (ΔOF1 ≤ 0 ) then
   update() /* accept */
else
  if (random() < e^(- ΔOF1 / T(i))) then
    update()  /* accept */
else
  reject_purturbation();
```

**Figure 5.** Outline of the simulated annealing algorithm.

## *Metropolis step and feasibility*

An iteration of the Metropolis step, *generate_function()*, consists of a perturbation operation, an accept/reject criterion, and a thermal equilibrium criterion. Perturbation in our strategy is performed randomly by selecting an event within a test case and substituting it with a randomly chosen event from the events set.

The acceptance criterion checks the change in $E$ due to the perturbation. If the change decreases the objective function, the perturbation is accepted and $C$ is updated. However, if the perturbation causes the objective function to increase, it is accepted only with a probability $e^{-\Delta OF1/T(i)}$. The main advantage of this Monte Carlo algorithm (Motwani and Raghavan 1995) is that the controlled uphill moves can prevent the system from being prematurely trapped in a bad local minimum-energy state. Note that for lower temperature values $T(i)$, the probability of accepting uphill moves becomes smaller; at very low (near-freezing) temperatures, uphill moves are no longer accepted. The perturbation–acceptance step is repeated

many times at every temperature after which thermal equilibrium is considered to be reached.

Perturbations can make $C$ infeasible if they violate the definition of continuity. But, the formulation of the energy function $E$ accounts for this infeasibility problem. The last term in $E$, $DC$, can be assigned a large weight, $\gamma$, so that infeasibility is severely penalized. Thus, infeasible test cases will be prevented at low temperatures.

### Cooling schedule

The cooling schedule is determined by running a heuristic algorithm that deduces the starting and freezing temperatures with respect to the number of uphill jumps. The initial temperature $T(0)$ is the temperature that yields a high initial acceptance probability of 0.93 for uphill moves. The freezing point is the temperature at which such a probability is very small (2–30), making uphill moves impossible and allowing only downhill moves. The cooling schedule used in this work is simple: $T(i + 1) = \theta \times T(i)$, with $\theta = 0.95$.

As the annealing algorithm searches the solution space, the best-so-far solution (with the smallest energy value) found is always saved. This guarantees that the output of the algorithm is the best solution it finds regardless of the temperature at which it terminates. Convergence is then detected when the algorithm does not improve on the best-so-far solution for a number of temperatures, say 20, in the colder part of the annealing schedule.

## Experimental results

In this section, we present the results of generating test cases using our proposed SO-SA algorithm and compare them with the results of incremental simulated annealing (INC-SA), greedy, and genetic algorithms (GA). Incremental SA generates test cases one at a time (rather than all tests simultaneously) containing a maximum of $K$ events at each iteration and adds the test case to the final configuration of the test suite. Incremental SA makes use of the same energy function. However, at the end of each iteration, the event frequency, coverage, and diversity matrices are saved, to be used by the energy function on the next iteration. The greedy algorithm accepts only the changes that decrease the value of the objective function, and do not allow uphill moves. It deals with the entire test suite over a number of iterations. The GA generates chromosomes, which are evaluated using the fitness function, for a number of generations using genetic operators (Alander 2008).

The four algorithms are applied on a state graph with 270 events as shown in Figure 6. We generated a test suite of 40 test cases, where each test case have a maximum of K test events. To simulate variant-length test case size,

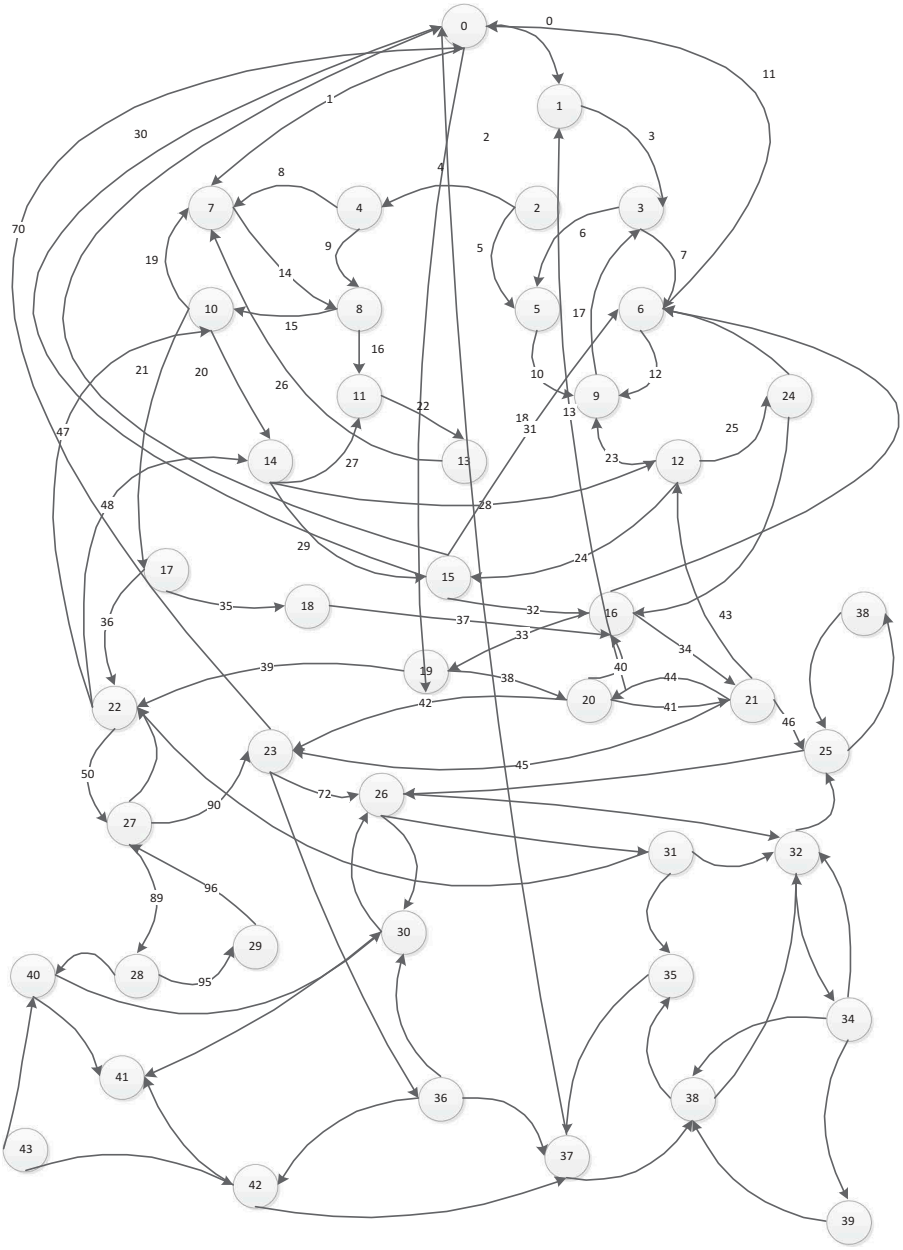


**Figure 6.** State graph of a web application.

we appended 10% fake edges ("No Edge") to the list of events. We ran the four algorithms with different test case sizes K = 10, 15, 20, and repeated the execution for each K value 10 times, for statistical purposes.

The SO-SA, incremental, and GA algorithms successfully generated 40 test cases, consisting of a continuous sequence of events. Close examination of the results reveals that the three algorithms successfully cover all the events in the

application. More importantly, they are able to generate a diversity suite that ensures testing the different parts of the Web 2.0 application. However, the performance of the SO-SA and GA algorithms are superior to the others, with SO-SA having a slight edge over the GA. This is due to yielding lower energy values when using the same test suite size with test cases of the same *K*. The greedy algorithm resulted in an unoptimized test suite. The greedy algorithm failed to generate continuous sequences of events in the test cases. The energy value converges fast within the initial iteration; however, no further improvements were obtained.

Tables 1–3 and Figures 7–9 show the comparison between the best, worst, and average energy function for the four strategies (SO-SA, incremental SA, greedy algorithm, and GA) obtained for different maximum event numbers in a specific test case. Table 4 depicts the standard deviation of the energy values.

The results show that SO-SA converged to the best energy values in all cases followed by GA. The greedy algorithm failed tremendously in comparison with the others. The results also show that the incremental SA limits its search by the early decisions on test case selection (made in early iterations) in contrast with the SO-SA that simultaneously generates test cases. Also, it shows that as the test case length, *K*, increases, the energy values for the four algorithms increase since it becomes harder to maintain relative diversity.

**Table 1.** Best final energy values of 10 runs for different test case sizes—*K*.

| Maximum number of events in test cases | Simultaneous-operation SO-SA | Incremental INC-SA | Greedy algorithm | Genetic algorithm |
|---|---|---|---|---|
| $K = 10$ | **1.214** | 1.353 | 31.000 | 1.268 |
| $K = 15$ | **1.729** | 1.819 | 56.160 | 1.828 |
| $K = 20$ | **2.194** | 2.413 | 83.320 | 2.732 |

**Table 2.** Worst final energy values of 10 runs for different test case sizes—*K*.

| Maximum number of events in test cases | Simultaneous-operation SO-SA | Incremental INC-SA | Greedy algorithm | Genetic algorithm |
|---|---|---|---|---|
| $K = 10$ | 1.382 | 1.968 | 37.478 | **1.374** |
| $K = 15$ | **1.828** | 2.503 | 68.600 | 1.924 |
| $K = 20$ | **2.834** | 2.860 | 93.791 | 3.240 |

**Table 3.** Average final energy values of 10 runs for different test case sizes—*K*.

| Maximum number of events in test cases | Simultaneous-operation SO-SA | Incremental INC-SA | Greedy algorithm | Genetic algorithm |
|---|---|---|---|---|
| $K = 10$ | **1.289** | 1.776 | 35.080 | 1.311 |
| $K = 15$ | **1.762** | 2.170 | 61.578 | 1.870 |
| $K = 20$ | **2.330** | 2.781 | 88.662 | 2.922 |

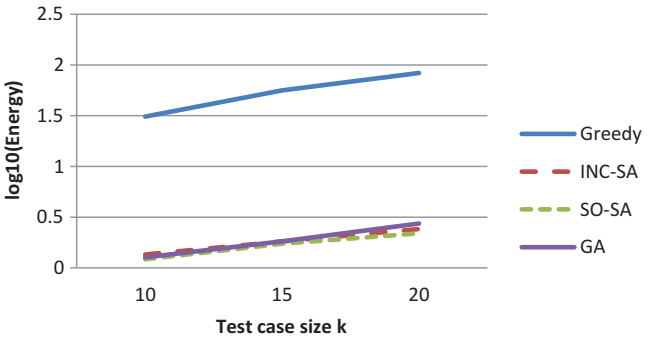


**Figure 7.** Best final energy values.

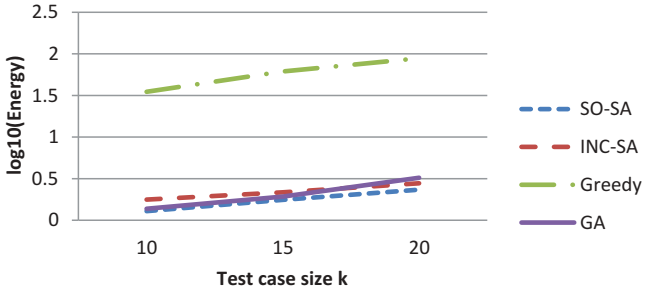


**Figure 8.** Worst final energy values.

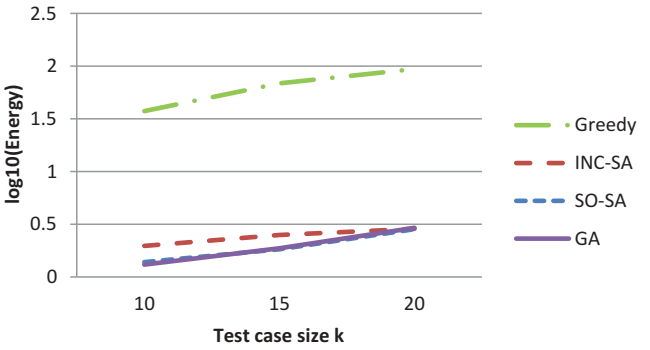


**Figure 9.** Average best energy values.

**Table 4.** Standard deviation of final energy values of 10 runs for different test case sizes—*K*.

| Maximum number of events in test cases | Simultaneous-operation SO-SA | Incremental INC-SA | Greedy algorithm | Genetic algorithm |
|---|---|---|---|---|
| $K = 10$ | 0.041 | 0.199 | 2.012 | **0.037** |
| $K = 15$ | **0.030** | 0.237 | 3.280 | 0.033 |
| $K = 20$ | **0.174** | 0.140 | 3.735 | 0.179 |

## Conclusion

We presented an optimization metaheuristic method for testing web applications. We also modeled the dynamic features of Web 2.0 using state transition diagrams. We used a SA algorithm to generate test cases as long sequences of semantically interacting events. Test cases were generated as sequences of semantically interacting events. We also formulated an energy function that is based on the capability of these test cases to provide high coverage of events, high diversity of events covered, and definite continuity of events. Our experimental results show that the proposed SO-SA algorithm yields better results than the incremental SA, competitive results with the GA, and significantly better results than a greedy algorithm. Future work aims to improve the components of the energy function based on larger applications, and to compare the SA algorithm with more metaheuristics.

## ACKNOWLEDGMENT

The three authors contributed equally to this paper. The work was partially supported by the Lebanese American University.

## ORCID

Ramzi A. Haraty http://orcid.org/0000-0002-6978-3627

## References

Alander, J. T. 2008. An indexed bibliography of genetic algorithms in testing. Technical Report 94-1-TEST, University of Vaasa, Vaasa, Finland.

Andrews, A., J. Offutt, and R. Alexander. 2005. Testing web applications by modelling with FSMs. *Software and System Modelling* 4 (3):326–345, July.

Benedikt, M., J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic Web sites. Accessed July 18, 2013. http://www2002.org/CDROM/alternate/654/.

Di Lucca, G. A., A. R. Fasolino, F. Faralli, and U. D. Carlini. 2002. Testing Web applications. Proceedings of the International Conference on Software Maintenance, Montreal, Canada, October. IEEE Computer Society.

Document Object Model (DOM). 2005. Accessed March 23, 2017. http://www.w3.org/DOM.

Elbaum, S., G. Rothermel, S. Karre, and M. Fisher. 2005. Leveraging user session data to support Web application testing. *IEEE Transactions of Software Engineering* 31 (3):187–202. doi:10.1109/TSE.2005.36.

Fejes, B. 2004. Test Web applications with HttpUnit". March 23, 2017. http://www.javaworld.com/javaworld/jw-04-2004/jw-0419-httpunit.html.

Ferguson, R., and B. Korel. 2006. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodologies* 5 (1):63–86. doi:10.1145/226155.226158.

Kirkpatrick, S., C. Gelatt, and M. Vecchi. 1983. Optimization by simulated annealing. *Science* 220:671–80. doi:10.1126/science.220.4598.671.

Kolawa, A., and D. Huizinga. 2007. *Automated defect prevention: Best practices in software management*. Hoboken, New Jersey, USA: Wiley-IEEE Computer Society Press. ISBN 0470042125.

Liu, C., D. Kung, P. Hsia, and C. Hsu. 2000. Structural testing of Web applications. Proceedings of the 11th IEEE International Symposium on Software Reliability Engineering, San Jose, CA, USA. 84–96, October.

Mansour, N., V. Isahakian, and I. Ghalayini. 2011. Scatter search technique for exam time-tabling. *Applied Intelligence* 34 (2):299–310. doi:10.1007/s10489-009-0196-5.

Mansour, N., H. Zeitunlian, and A. Tarhini. 2013. Optimization metaheuristic for software testing. In EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation II. Advances in Intelligent Systems and Computing, ed. by : Schütze O. et al., Vol 175, Berlin, Heidelberg: Springer.

Marchetto, A., P. Tonella, and F. Ricca. 2008. State-based testing of AJAX Web applications. Proceedings of IEEE International Conference on Software Testing (ICST), Lillehammer, Norway, April.

Marchetto, A., P. Tonella, and F. Ricca. 2009. Search-based testing of AJAX web applications. Proceedings of IEEE Search Based Software Engineering, Windsor, UK, May.

Memon, M., M. Pollack, and L. Soffa. 2001. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering* 27 (2):144–55. doi:10.1109/32.908959.

Motwani, R., and P. Raghavan. 1995. *Randomized algorithms*. New York: Cambridge University Press. ISBN 0-521-47465-5.

Nikolik, B. 2006. Test diversity. *Information and Software Technology* 48:1083–94. doi:10.1016/j.infsof.2006.02.001.

O'Reilly, T. 2005. Design patterns and business models for the next generation of software. Accessed March 23, 2017. http://oreilly.com/web2/archive/what-is-web-20.html.

Petrenko, A., S. Boroday, and R. Groz. 2004. Confirming Configurations in EFSM Testing. *Software Engineering, IEEE Transactions* 30:29–42. doi:10.1109/TSE.2004.1265734.

Ricca, F., and P. Tonella. 2001. Analysis and testing of Web applications. Proceedings of the International Conference on Software Engineering, Toronto, Ontario, Canada. 25–34, May.

Szalvay, V. 2004. *An introduction to agile software development*. Wien, Austria: Danube Technologies Inc.

Tarhini., A., N. Mansour, and H. Fouchal. 2010. Testing and regression testing for Web services based applications. *International Journal of Computing and Information Technology* 2 (2):195–217.

Web Application Testing Tools. 2001. Accessed July 18, 2013. http://logitest.sourceforge.net/logitest/index.html.

Zeitunlian, H. 2012. Metaheuristic Algorithm for Testing Web 2.0 Applications. Master's Thesis. Lebanese American University.