Building Rich Internet Applications Models: Example of a Better Strategy

Suryakant Choudhary[1], Mustafa Emre Dincturk[1], Seyed M. Mirtaheri[1], Guy-Vincent Jourdan[1,2], Gregor v. Bochmann[1,2], and Iosif Viorel Onut[3,4]

[1] EECS - University of Ottawa
[2] Fellow of IBM Canada CAS Research, Canada
[3] Research and Development, IBM® Security AppScan®, Security Systems
[4] IBM Canada Software Lab, Canada
{schou062,mdinc075,smirt016}@uottawa.ca,
{gvj,bochmann}@eecs.uottawa.ca,
vioonut@ca.ibm.com

Abstract. Crawling "classical" web applications is a problem that has been addressed more than a decode ago. Efficient crawling of web applications that use advanced technologies such as AJAX (called Rich Internet Applications, RIAs) is still an open problem. Crawling is important not only for indexing content, but also for building models of the applications, which is necessary for automated testing, automated security and accessibility assessments and in general for using software engineering tools. This paper presents a new strategy to crawl RIAs. It uses the concept of Model-Based Crawling (MBC) first introduced in [1], and introduces a new model, the "menu model", which we show to be much simpler than previous models for MBC and more effective at building models than previously published methods. This method and others are compared against a set of experimental and real RIAs.

Keywords: Crawling, RIAs, AJAX, Modeling.

1 Introduction

The ability to automatically extract a model of a website is important for several reasons. The most obvious one is to index the content of the sites, which is done through "crawling". Indexing is obviously a central feature of the Web, but not the only reason why inferring models is important. We also require models for tasks related to good software engineering: models are needed as input for automated testing of applications ("model-based testing"), models are also needed for automated security assessments, for automated usability assessments, or simply as a way to better understand the structure of the website.

Nearly two and a half decades of research in the area of model extraction and crawling has produced a large body of work with many powerful solutions [2]. The majority of the studies, however, focus on traditional web applications, where the HTML view of the page is generated on the server side. In this model, there is a one-to-one relation between the URL of the page and the state of its *Document Object Model* (DOM) [3]. Thus, many of the proposed web crawlers use the URL to identify the state of the DOM. Such assumption reduces the basic task of crawling the Web to the task of finding all the valid and reachable URLs from a set of seed URLs.

However, the so-called *Rich Internet Applications* (RIAs) break the one-to-one relationship between the URL and the state of the DOM. In RIAs, DOMs are partially updated by client-side script execution (such as JavaScript®), and asynchronous calls to the server are done through technologies such as AJAX [4]. Such sophisticated client-side applications create a one-to-many relation between the URL and the reachable DOM states associated with that URL.

This evolution is positive, but comes at a cost which has been underestimated: the crawling techniques developed for traditional web applications just do not work on RIAs. We have lost our ability to crawl and model web applications as they are typically created today[1]. Even simple websites are not immune to the problem since common tools to create and maintain website content are increasingly adding AJAX-like scripts to the page. We need to address this issue, which means to develop web crawlers that do not rely solely on the URL to uniquely identify the state of the application, but also take into consideration the DOM structure and its properties to identify different states of the application. There is some work being done in that domain (see [5] for an overview), but more must be done. This paper is one step in this direction.

Crawling RIAs is much more complex than crawling traditional web applications. The one-to-many relation between a URL and states of the DOM can be modeled as a directed graph referred to as the *application graph*. In the application graph, each state of the DOM is a node, and each JavaScript event is a directed edge. To construct such a graph, one must differentiate between different states of the DOM, which is a challenge in itself, but outside the scope of this paper (see e.g. [6] for a discussion on the topic). In this model, taking an edge from a node means executing a JavaScript event from the DOM that the node represents.

After defining the application graph, the task of crawling a RIA is reduced to the task of discovering every state in the application graph. The state that is reached when a given URL is loaded is called the "initial state of the URL". For a crawler to ensure that all states reachable from a given URL are discovered, the crawler has to start from the initial state of the URL, take every possible transition, and do this for every newly discovered state recursively. This often takes a long time. It is thus interesting for a crawler to discover as many states as possible during early stages of the crawl, and postpone executing events that most probably lead to visited states.

To this end, we have introduced a general approach called *model-based* crawling [1], where a crawling strategy aims at discovering the states of the application

[1] See e.g. `https://developers.google.com/webmasters/ajax-crawling/docs/getting-started`, in which Google suggests to create static URLs to index the pages that will not be reached by the crawler because of AJAX.

as soon as possible by making predictions based on an anticipated model for the application. In this paper, we propose a new strategy, called the Menu strategy, using the model-based crawling approach. This new algorithm is simpler and more efficient to discover all reachable DOM states in a RIA than the other known strategies.

The rest of this paper is organized as follows: in Section 2, we give an overview of model-based crawling. In Section 3, we explain the proposed strategy in details. Section 4 presents the experimental study. In Section 5, a summary of related works is presented. In Section 6, we conclude the paper.

2 Overview

Building a model of a RIA is potentially a very lengthy process, because of the large number of states and transitions involved. Because of this, many of the existing strategies do not try to build a complete model of the application being crawled. Our approach is different: we insist that under some assumptions, given enough time the strategy should produce a complete model of the RIA. On the other hand, we acknowledge that, most of the time, we will not have enough time to complete the crawl. Thus, our first goal is to produce a complete model as efficiently as possible, which means that we want to minimize the number of events we need to execute to produce such a model. Our second goal is that, as we produce this model, we should discover as many states as possible, as early as possible during the crawl. Because, in most cases, it is more important to find the states than it is to find the transitions. If we are not going to run the crawl to the end, we want to ensure that the partial model being built will contain as much states as possible.

When we crawl a website, we make the following assumptions: we assume that, if user inputs are involved, we have access to a collection of sample inputs that are good enough to build the model. We do not address here the question of how to generate such inputs. The second assumption is that the RIA being crawled is deterministic from the point of view of the crawler. This means that, from the same state, the same action will always produce the same result (go to the same state). Although this assumption is fairly commonly made in the literature, we recognize that it is a very limiting assumption and that more work will have to be done to relax it in the future. Finally, we assume that we can always "reset" the RIA by reloading the URL, and thus the underlying application graph is strongly connected.

In general, it is not possible to devise a strategy that would be efficient at finding the states early, since the underlying graph could be any graph. We have introduced model-based crawling as a solution to this problem [1]. With model-based crawling, we initially assume that application will follow a particular behavioral model referred to as *meta-model*. It is anticipated that the model of the application will be an instance of this meta-model. An efficient (ideally, optimal) strategy is designed based on this anticipation. However, it is not strictly assumed that the RIA being crawled will actually follow the meta-model. During the crawl, each time we see a difference between the anticipated behavior and the actual behavior, we adapt the strategy accordingly.

A model-based strategy usually consists of two phases:

1. **State exploration phase** where the objective is to discover all the application states as predicted by the meta-model of the strategy.
2. **Transition exploration phase** where the objective is to execute all remaining events, to complete the model.

It is possible that, during the second phase, new states are discovered, in which case we will switch back to the first phase. Thus, a model-based crawling strategy may alternate between these two phases multiple times before finishing the crawl. The strategy finishes the crawl when it has executed all the events in the application, which guarantees to have discovered all the states of the application.

The first model-based crawling strategy is the "Hypercube" strategy where the application is anticipated to have a hypercube structure [1]. The Hypercube strategy is an optimal strategy for the RIAs that fully follow the hypercube meta-model. However, in practice, few RIAs follow this model, and the algorithms involved are rather complex. Even though the results were better than other strategies even for RIAs that do not follow this model, we present here a new strategy that is better still, much easier to understand and is based on a meta-model more commonly found in RIAs.

3 Menu Model

The proposed crawling strategy is based on the idea that some events will always lead the application to the same resulting state, regardless of the source state from which the event is executed. These kind of events are referred to as the "menu events".

We called this new model *menu model* because our menu events are often the intended model behind application menus. Such behavior is realized by the menu items present in a web application such as *home, help, about us* etc.

Once an event is identified as a menu event, we can use it to anticipate some part of the application graph, and use this anticipated graph to build an efficient strategy. Thus, the core of the strategy is to identify these menu events, and then execute the events that are not menu events sooner than the menu events (since menu events are anticipated to produce known states). In practice, we prioritize the events based on the execution history:

1. **Globally unexecuted events**: This category represents the events that have not yet been executed at any state discovered so far. Events in this category have the highest priority.
2. **Locally unexecuted events**: This category represents the events that have been executed at some discovered state but have not been executed at the current state of the application. Events in this category are further divided into the following subcategories:

(a) **Non-classified events**: Events in this subcategory has been executed only once globally. A second execution is necessary to classify the event. Events in this subcategory have the second highest priority next to the globally unexecuted events.
(b) **Menu events**: Events in this subcategory follow the menu model hypothesis when the first two executions are considered: their executions from two different states have led to the same state. They have the lowest priority.
(c) **Self-Loop events**: Events in this subcategory have not changed the state of the application in their first two executions. They have the same priority as the menu events.
(d) **Other events**: All the remaining events belong to this category. These are the events that have shown neither menu nor self-loop behavior in their first two executions. These events have the same priority as non-classified events.

Since the events in the menu and self-loop categories are not expected to lead to a new state, they have the lowest priority.

The categorization of the events is done throughout the crawl. The priority sets are updated as new events are found in newly discovered states and as more information about results of the execution instances of the events become available.

3.1 State Exploration Phase

The primary goal of the state exploration phase is to discover all the states of the application as soon as possible. To do so, the strategy constructs and maintains a graph model of the application. The application graph is a weighted directed graph, $G = (V, E)$ where V represents the states discovered and E represents the edges. An edge may be an executed event, a *reset*, or a *predicted* transition. A reset is the action of resetting the application to its initial state by reloading the URL. For simplicity, we assume each event to have the same unit cost, but the cost of reset is different and it depends on the application being crawled. A predicted edge corresponds to a non-classified event or a menu event that is not executed in the source state (for the purpose of predicting transitions, all non-classified events are assumed to be menu events). In the case of a non-classified event, the predicted resulting state is the state which was reached on the first execution of the event, and in the case of a menu event it is the resulting state of the menu event. A self-loop predicted edge correspond to an unexecuted self-loop event. In this case, the predicted resulting state is the starting state of the self-loop edge. Figure 1 shows an instance of G.

The state exploration phase starts by categorizing the events (initially, the crawler only knows the events on the initial state; but, as the crawl progresses, previously unseen events can be found on newly discovered states). Each event initially belongs to the globally unexecuted category. Unexecuted events are then picked according to the priority sets. All the instances of the events from a higher

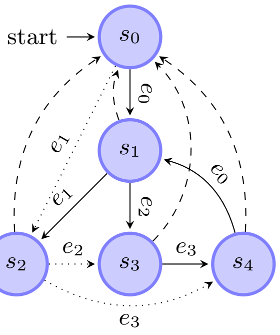

Fig. 1. An example of application graph G under construction: solid lines are executed transitions, dashed lines are resets, dotted lines are predicted transitions

priority set are exhausted before executing an event from a lower priority set. Among the events with the same priority, the priority is given to any event which is closer to the current state than the others (closeness is in terms of number of transitions that needs to be taken to reach a state where the event is enabled and unexecuted), otherwise one is chosen at random. During the state exploration phase, we execute all the unexecuted events in the application, except for categorized menu and self-loop events.

Once an event is picked for execution, the strategy always uses the shortest known path from the current state s_{curr} to the state s_{next} where the event is going to be executed. This calculated shortest path may contain predicted transitions. A predicted transition may of course be wrong, and the application may end up in a state that is not the predicted one. During the execution of the path, the strategy verifies, after each predicted transition, that the state reached is the one predicted. When this is not the case, the crawled RIA contradicts the menu model (at least from that state, and for this event). To adapt to such a violation, the strategy discards the current path and looks for the next unexecuted event from the state reached.

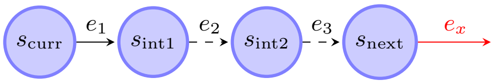

Fig. 2. Path from the current state to state s_{next} where the next event can be executed. Solid lines represent known transitions, and dotted lines represent predicted transitions.

For instance, considering the execution of the example path shown in Figure 2, let us assume that there is a violation when the predicted transition e_2 (originating from s_{int1}) is taken. As Figure 3 shows, after executing event e_2 on s_{int1}, we reach state s' instead of s_{int2}. Due to this violation, the menu strategy ignores the rest of the path segments, and builds a new path from the current state (s') to a next state with an unexecuted event.

During the execution of a path, each predicted transition leads to the execution of an event that had not been executed from that state before, which permits the categorization of that event if it is not already categorized.

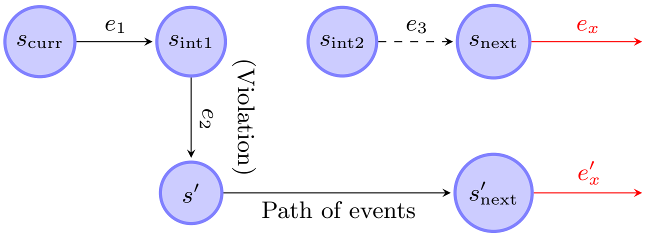

Fig. 3. Example of a violation for the path in Figure 2

3.2 Transition Exploration Phase

The state exploration phase executes all the events discovered during that phase, except for the events in the menu and the self-loop categories. Once the state exploration phase is over, the menu strategy moves to the transition exploration phase. The transition exploration phase verifies the validity of the assumptions made at the state exploration phase by executing all these remaining events. In an application that follows the menu model, all the states of the application are found by the end of the state exploration phase. Any violating menu or self-loop events, however, may lead to the discovery of a new state.

During the transition exploration phase, the strategy tries to find the least costly path to execute all the remaining events in the application. The cost of this path is measured in terms of the total number of events and resets required.

If we define *a walk of the graph* as a sequence of adjacent edges, the transition exploration problem can be mapped to the problem of finding the least costly walk of the graph that traverses all the edges representing the unexecuted events at least once. During the transition exploration phase, should the execution of any unexecuted event lead to the discovery of a new state, the strategy switches back to the state exploration phase. This mechanism expedites finding new states. Thus, the menu strategy might alternate between the state and transition exploration phases many times before it finishes the crawl of the application.

Graph Walk. The transition exploration phase uses a walk generator function to calculate a walk that covers all of the unexecuted events. During the calculation of the graph walk, the application graph includes predicted transitions.

Hence, executing the event sequence in the walk might not result in the expected state. In fact, a single violation can make the event sequence invalid. To avoid this, a step-wise approach in construction of the whole walk is taken. The walk generator function splits the event sequence into multiple walk segments. Each walk segment may start with a reset, may be followed by zero or more already executed events, and ends with an unexecuted event.

Considering the example in Figure 1 where the results of all the unexecuted events have been assumed, a possible walk that covers every unexecuted event is the sequence $< e_1, e_3, e_0, e_1, e_2 >$, which starts at the initial state, s_0, and terminates at s_3.

Our immediate situation is similar to the problem known as the Rural Chinese Postman Problem (RCPP) [7], where given a graph we want a least cost tour covering only a subset of the edges. The application graph contains known transitions corresponding to executed events and predicted transitions corresponding to unexecuted menu and self-loop events. We need a least cost tour to execute all the remaining unexecuted events.

Unfortunately, the RCPP is an NP-complete problem, so we do not attempt to solve this problem. Instead, we use the Chinese Postman Problem (CPP). In CPP, given a graph we want a least cost tour of all the edges. Unlike the RCPP, there are polynomial algorithms for the CPP. However, this is not a perfect analogy to our situation: in the current graph, we have both executed and predicted transitions, and we only want to execute the predicted ones. If we consider the subgraph containing only the predicted transitions, this subgraph may not be connected, and a tour may not exist. To address this problem, we augment this subgraph with a few of the known transitions (including resets if necessary), until the graph is strongly connected again. We then use CPP to create a tour that goes over every transitions. This solution gives reasonably good results (although clearly non optimal) at a small computational cost.

Violation and Strategy Adaptation. When going over the tour, each predicted transition may lead to a violation of the assumption, and the application can end up in a state that is not the one predicted. There are two cases to handle:

1. **Wrong known state:** This is the case where the resulting state has been discovered previously, but it is not the expected state. When this happens, the predicted edge is removed from the graph, replaced with the newly executed transition. At this point, we end up in the wrong state in the tour. Instead of recomputing a tour, we have opted for a simpler solution: the strategy keeps the original walk, and brings the application back to the state that was expected to be reached initially. To do this, we simply find the shortest known path that does not contain any predicted edges from the current state to that next state, and execute it first.
2. **New state:** Should a violation lead to the discovery of a new state, the crawling strategy switches back immediately to the state exploration phase. However, we do not discard the calculated CPP walk, which is reused later when the strategy reaches the transition exploration phase again. At this

point, the existing CPP is augmented to include any additional discovered unexecuted events.

Due to space constraints, we do not include more details which can be found in [8].

4 Implementation and Evaluation

In this section, we present our experimental results, comparing the efficiency of the Menu strategy against many other existing crawling strategies on several AJAX-based RIAs.

4.1 Measuring the Efficiency of a Strategy

As explained before, our definition of an efficient strategy is a strategy that builds the entire model quickly, while finding all the states as early as possible in the process. In order to measure speed, instead of measuring time, we measure the number of event executions and the number of resets required by each strategy to complete both tasks (find all the states, find the complete model). This is reasonable since the time spent for event executions and resets dominates the crawling time and the numbers depend only on the decisions of the strategy. And this way, the results do not depend on the hardware that is used to run the experiments and are not affected by the network delays which can vary in different runs.

We combine these numbers to define a cost unit as follows. We measure for each application the following two values. $t(e)_{avg}$: the average event execution time obtained by measuring the time for executing each event in a randomly selected set of events in the application and taking the average, and $t(r)_{avg}$: the average time to perform a reset. For simplicity, we consider each event execution to take $t(e)_{avg}$ and take this as a cost unit. Then, we calculate "the cost of reset": $c_r = t(r)_{avg}/t(e)_{avg}$. Finally, the cost that is spent by a strategy to find all the states of an application is calculated by $n_e + n_r \times c_r$ where n_e and n_r are the total number of events executed and resets used by the strategy to find all the states, respectively[2].

4.2 Crawling Strategies Used for Comparison

- Optimized Standard Crawling Strategies: The standard crawling strategies are Breadth-First and Depth-First. We use "optimized" versions of these strategies, meaning that when there is a need to move from the current state to another known state, the shortest known path from the current state to

[2] We measure the value of c_r before crawling an application and give this value as a parameter to each strategy. A strategy, knowing how costly a reset is compared to an average event execution, can decide whether to reset or not when moving from current state to another known state.

the desired state is used. This is in contrast to using systematic resets. The results presented here with the optimized versions are much better than the ones obtained using the standard, non-optimized Breadth-First and Depth-First strategies.
- Greedy Strategy [9]: This is a simple strategy that prefers to explore an event from the current state, if there is one. Otherwise, it chooses an event from a state that is closest to the current state.
- Other Model-based Crawling Strategies: We also compare with other existing model-based strategies: The Hypercube strategy [1] is based on the anticipation that the application has a hypercube model. The Probability Strategy [10] prioritizes the events by estimating their probabilities of discovering a new state based on their previous explorations.
- The Optimal Cost: We also present the optimal cost of discovering all the states for each application. This cost is calculated once the model is known (after the application is crawled first with one of the strategies). Finding an optimal path that visits every state in a known model is possible by solving an Asymmetric Traveling Salesman Problem (ATSP). We use an exact ATSP solver [11] to find this path. This gives us an idea of how far from the optimal speed each strategy is (for the first phase, find all the states). Of course, this optimal is not a strategy on its own, and can only be calculated once the entire model is known.

4.3 Subject Applications

We are comparing the strategies using two test RIAs and four real RIAs[3]. This number is not as large as we would like, but we are limited by the tools that are available to us. Each new RIA requires a significant amount of work before we can crawl it[4]. With the increasing exposure to this problem, better tools will be made available, and we will be able to test our solutions on a much broader test set.

- Bebop: This is an AJAX-based interface to browse a list of publications. We have used an instance that contains 5 publications. It has 1,800 states and 145,811 transitions. The measured cost of reset is 2.

[3] `http://ssrg.eecs.uottawa.ca/testbeds.html`

[4] We stress that the work in question is not related to the strategy described here, but to the limitation of the available tools. One approach to implement a RIA crawler is to control an external browser using an API such as Selenium WebDriver (as Crawljax [12] does). The main drawback of this approach is the inability to detect automatically all the events in a page since the DOM interface does not have a method to check if an element has an event registered dynamically (using `addEventListener` method in JavaScript). So, the user needs to specify the elements that should be interacted with in an application. Our approach is to implement a browser as part of the crawler. Thus, our crawler has more control over the application and can detect automatically all the events in a page. However, this requires more work since we need to make sure that our browser supports all the functionality required by the RIA.

– jQuery FileTree: This is an AJAX-based file explorer. For this study, we used an instance that allows browsing Python source code. It has 214 states, 8,428 transitions. The measured cost of reset is 2.
– Periodic Table: This is an AJAX-based periodic table. It has 240 states, 29,034 transitions. The measured cost of reset is 8.
– Clipmarks: This was a AJAX-based social network. We have used a partial local copy of this website for the experimental study. It has 129 states, 10,580 transitions. The measured cost of reset is 18.
– Altoro Mutual: This is an AJAX version of a demo website in the form of a fictional banking site. It has 45 states, 1,210 transitions. The measured cost of reset is 2.
– TestRIA: This is a AJAX test application in the form of a generic homepage. It has 39 states, 305 transitions. The measured cost of reset is 2.

4.4 Experimental Setup

We have implemented all the mentioned crawling strategies in a prototype of IBM® Security AppScan® Enterprise[5]. Each strategy is implemented as a separate class in the same code base, so they use the same DOM equivalence mechanism, the same event identification mechanism, and the same embedded browser. For this reason, each strategy extracts the same model for an application.

We crawl each application with each strategy ten times and present the average of these crawls. In each crawl, the events of each state are randomly shuffled before they are passed to the strategy. The aim here is to eliminate influence caused by exploring the events of a state in a certain order since the strategy may not define an exploration priority for the events on a state.

4.5 Costs of Discovering States (Strategy Efficiency)

The box plots in Figure 4 show the results. For each application and for each strategy, the figure contains a box plot. A box plot consists of a line and a box on the line. The minimum point of the line shows the cost of discovering the first state (always equal to the cost of reset for the application). The lower edge, the line in the middle and the higher edge of the box show the cost of discovering 25%, 50% and 75% of the states, respectively. The maximum point of the line shows the cost of discovering all the states. The plots are drawn in logarithmic scale for better visualization. Each horizontal dotted line shows the optimal cost for the corresponding application.

The results show that for all applications the Greedy strategy and the model-based strategies are significantly more efficient than Breadth-First and Depth-First. It can also be seen that the Menu strategy has the best performance to discover all the states except for the Bebop where it is very close to the best. In

[5] Details are available at `http://ssrg.eecs.uottawa.ca/docs/prototype.pdf` Since our crawler is built on top of the architecture of a commercial product, we are not able to provide open-source implementations of the strategies currently.

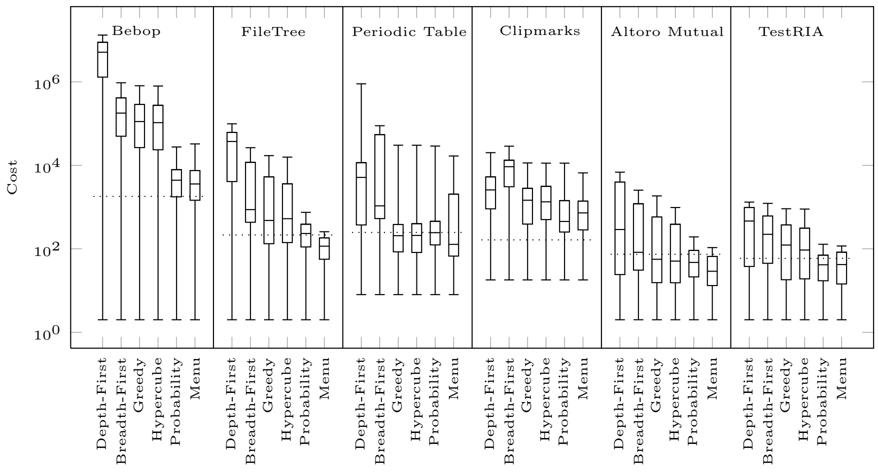

Fig. 4. Costs of Discovering the States (Strategy Efficiency), in logarithmic scale. Each horizontal dotted line shows the optimal cost for the corresponding application.

4 out of 6 cases, it was the first to discover the 75% of the states. In addition, the Menu strategy was the first to discover the 50% and the 25% of the states in all cases, except for Clipmarks where it is very close to the best. This is particularly important if one assumes that the crawl will not be run to the end and that in most cases it will be cut short. It shows that the Menu is the strategy that will provide the most information after the least amount of time.

4.6 Costs of Complete Crawl

The previous results show the costs of discovering all the states. However, the crawl does not end at this point since a crawler cannot know all states are discovered until all the events are explored from each state (in other words, we could provide this information only because we have run the tests to the end). In Table 1, the total number of events and the total number of resets during the crawl are shown as well as the costs calculated based on these numbers.

It can be seen that the model-based strategies and the Greedy strategy finish crawling with a significantly less cost compared with Breadth-First and Depth-First. The Menu is in the same ballpark as the other model-based strategies, but not better. However, the complete crawl is not as important a factor as finding all the states, as explained before.

5 Related Works

A survey of traditional crawling techniques is presented in [2]. For RIA crawling, a recent survey is presented in [5]. Except for [1,10,13] which present other

Table 1. Total Costs of Crawling

	Bebop			FileTree			Periodic Table			Clipmarks			Altoro Mutual			TestRIA		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	13,386,210	27	13,386,264	99,336	13	99,362	897,358	236	899,246	19,569	72	20,868	6,876	34	6,944	1,433	1	1,435
Breadth-First	943,001	8,732	960,466	26,375	1,639	29,652	64,850	14,633	181,916	15,342	926	32,015	3,074	334	3,742	1,216	55	1,326
Greedy	826,914	27	826,968	20,721	13	20,747	29,926	236	31,814	11,396	56	12,398	2,508	34	2,576	1,001	1	1,003
Hypercube	816,142	27	816,196	19,865	13	19,891	29,921	236	31,809	11,350	56	12,356	2,489	34	2,557	994	1	996
Probability	816,922	27	816,976	19,331	13	19,357	29,548	236	31,436	11,456	62	12,563	2,451	34	2,520	972	1	974
Menu	814,220	27	814,274	19,708	13	19,734	37,489	236	39,377	11,769	71	13,043	2,457	35	2,527	974	1	976

model-based crawling strategies and [9] which presents the Greedy strategy, the published research uses Breadth-First or Depth-First strategies for crawling RIAs. As we have seen, Breadth-First and Depth-First strategies are less efficient than the Greedy and the model-based strategies.

[14] and [15] suggest algorithms to index a RIA. [15] offers an early attempt in crawling AJAX applications based on user events and building the model of the application. The application model is constructed as a graph using the Breadth-First strategy. [14] introduces an AJAX-aware search engine for indexing the contents of RIAs. In this model components are adapted to handle RIAs. The crawler identifies JavaScript events and runs a standard Breadth-First search on them. [16] offers an algorithm, called *AjaxRank*, similar to *PageRank* [17] tailored to RIAs, to give weight to different states based on the connectivity.

[18–20] seek to automate regression and other testing of a RIA. *Crawljax* [12, 21] constructs a state-flow graph of the application by exercising client-side code and identifying the events that change the state of the application. Crawljax differentiates states using Levenshtein distance method [22], and uses a Depth-First strategy. [23] describes the derivation of test sequences from the application model obtained by crawling. [24] is similar, but takes a white-box testing approach where the program fragments of the states are analyzed.

Several other tools exist to create an FSM model of the application. *RE-RIA* [25] uses execution traces to create the FSM model of the application. As an improvement to *RE-RIA*, *CrawlRIA* [26] generates the execution traces by running a Depth-First strategy. *CreRIA* facilitate reverse engineering of a RIA for dynamic analysis. *DynaRIA* offers a tool to comprehend a RIA better for testing and other purposes. It also helps to visualize the run-time behavior of the application.

6 Conclusion

A new model-based crawling algorithm was introduced: the Menu model. The proposed architecture models the web application based on the JavaScript events in each state of the DOM. It makes assumptions about the category of events in order to derive a strategy, then learns, and adapt its categories as the crawling proceeds. A prototype of the system is implemented and the results are evaluated against several other model-based crawling algorithms. We have shown empirically that Menu strategy is better than other known strategies when it comes to finding all the states of the application being modeled.

Acknowledgments. This work is partially supported by the IBM Center for Advanced Studies, and the Natural Sciences and Engineering Research Council of Canada (NSERC).

The views expressed in this article are the sole responsibility of the authors and do not necessarily reflect those of IBM.

References

1. Benjamin, K., von Bochmann, G., Dincturk, M.E., Jourdan, G.-V., Onut, I.V.: A strategy for efficient crawling of rich internet applications. In: Auer, S., Díaz, O., Papadopoulos, G.A. (eds.) ICWE 2011. LNCS, vol. 6757, pp. 74–89. Springer, Heidelberg (2011)
2. Olston, C., Najork, M.: Web crawling. Found. Trends Inf. Retr. 4(3), 175–246 (2010)
3. World Wide Web Consortium (W3C): Document Object Model (DOM) (2005), `http://www.w3.org/DOM/`
4. Garrett, J.J.: Ajax: A new approach to web applications (2005), `http://www.adaptivepath.com/publications/essays/archives/000385.php`
5. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Moosavi, A., von Bochmann, G., Jourdan, G.V., Onut, I.V.: Crawling rich internet applications: the state of the art. In: Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research, CASCON 2012, pp. 146–160 (2012)
6. Choudhary, S., Dincturk, M.E., Bochmann, G.V., Jourdan, G.V., Onut, I.V., Ionescu, P.: Solving some modeling challenges when testing rich internet applications for security. In: 2012 International Conference on Software Testing, Verification, and Validation, pp. 850–857 (2012)
7. Eiselt, H.A., Gendreau, M., Laporte, G.: Arc routing problems, part ii: The rural postman problem. Operations Research 43(3), 399–414 (1995)
8. Choudhary, S.: M-crawler: Crawling rich internet applications using menu metamodel. Master's thesis, EECS - University of Ottawa (2012), `http://ssrg.site.uottawa.ca/docs/Surya-Thesis.pdf`
9. Peng, Z., He, N., Jiang, C., Li, Z., Xu, L., Li, Y., Ren, Y.: Graph-based ajax crawl: Mining data from rich internet applications. In: 2012 International Conference on Computer Science and Electronics Engineering (ICCSEE), vol. 3, pp. 590–594 (March 2012)
10. Dincturk, M.E., Choudhary, S., von Bochmann, G., Jourdan, G.-V., Onut, I.V.: A statistical approach for efficient crawling of rich internet applications. In: Brambilla, M., Tokuda, T., Tolksdorf, R. (eds.) ICWE 2012. LNCS, vol. 7387, pp. 362–369. Springer, Heidelberg (2012)
11. Carpaneto, G., Dell'Amico, M., Toth, P.: Exact solution of large-scale, asymmetric traveling salesman problems. ACM Trans. Math. Softw. 21(4), 394–409 (1995)

12. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. TWEB 6(1), 3 (2012)
13. Benjamin, K., Bochmann, G.V., Jourdan, G.V., Onut, I.V.: Some modeling challenges when testing rich internet applications for security. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2010, pp. 403–409. IEEE Computer Society, Washington, DC (2010)
14. Duda, C., Frey, G., Kossmann, D., Zhou, C.: Ajaxsearch: crawling, indexing and searching web 2.0 applications. Proc. VLDB Endow. 1(2), 1440–1443 (2008)
15. Duda, C., Frey, G., Kossmann, D., Matter, R., Zhou, C.: Ajax crawl: Making ajax applications searchable. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE 2009, pp. 78–89. IEEE Computer Society, Washington, DC (2009)
16. Frey, G.: Indexing ajax web applications. Master's thesis, ETH Zurich (2007), `http://e-collection.library.ethz.ch/eserv/eth:30111/eth-30111-01.pdf`
17. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web, Standford University, Technical Report (1998)
18. Roest, D., Mesbah, A., van Deursen, A.: Regression testing ajax applications: Coping with dynamism. In: ICST, pp. 127–136. IEEE Computer Society (2010)
19. Bezemer, C.P., Mesbah, A., van Deursen, A.: Automated security testing of web widget interactions. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE 2009, pp. 81–90. ACM, New York (2009)
20. Mesbah, A., van Deursen, A.: Invariant-based automatic testing of ajax user interfaces. In: IEEE 31st International Conference on Software Engineering, ICSE 2009, pp. 210–220 (May 2009)
21. Mesbah, A., Bozdag, E., Deursen, A.V.: Crawling ajax by inferring user interface state changes. In: Proceedings of the 2008 Eighth International Conference on Web Engineering, ICWE 2008, pp. 122–134. IEEE Computer Society, Washington, DC (2008)
22. Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady 10, 707 (1966)
23. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of ajax web applications. In: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST 2008, pp. 121–130. IEEE Computer Society, Washington, DC (2008)
24. Artzi, S., Dolby, J., Jensen, S.H., Møller, A., Tip, F.: A framework for automated testing of JavaScript web applications. In: Proc. 33rd International Conference on Software Engineering (ICSE) (May 2011)
25. Amalfitano, D., Fasolino, A.R., Tramontana, P.: Reverse engineering finite state machines from rich internet applications. In: Proceedings of the 2008 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 69–73. IEEE Computer Society, Washington, DC (2008)
26. Amalfitano, D., Fasolino, A.R., Tramontana, P.: Rich internet application testing using execution trace data. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2010, pp. 274–283. IEEE Computer Society, Washington, DC (2010)