A Statistical Approach for Efficient Crawling of Rich Internet Applications

Mustafa Emre Dincturk[1,3], Suryakant Choudhary[1,3], Gregor von Bochmann[1,3],
Guy-Vincent Jourdan[1,3], Iosif Viorel Onut[2]

[1] EECS, University of Ottawa. 800 King Edward Avenue,
K1N 6N5, Ottawa, ON, Canada
[2] Research and Development, IBM® Security AppScan® Enterprise, IBM, 1 Hines Rd, Ottawa, ON, Canada
[3] IBM Canada CAS Research
{ mdinc075, schou062}@uottawa.ca
{bochmann, gvj}@eecs.uottawa.ca
vioonut@ca.ibm.com

Abstract. Modern web technologies, like AJAX result in more responsive and usable web applications, sometimes called Rich Internet Applications (RIAs). Traditional crawling techniques are not sufficient for crawling RIAs. We present a new strategy for crawling RIAs. This new strategy is designed based on the concept of "Model-Based Crawling" introduced in [3] and uses statistics accumulated during the crawl to select what to explore next with a high probability of uncovering some new information. The performance of our strategy is compared with our previous strategy, as well as the classical Breadth-First and Depth-First on two real RIAs and two test RIAs. The results show this new strategy is significantly better than the Breadth-First and the Depth-First strategies (which are widely used to crawl RIAs), and outperforms our previous strategy while being much simpler to implement.

Keywords: Rich Internet Applications, Web Crawling, Web Application Modeling.

1 Introduction

Web applications have been undergoing a significant change in the past decade. Initialy, the web applications were built using simple HTML pages on the client side. Each page had a unique URL to access it. The client (web browser) would send a request for these URLs to the server which in turn would send the corresponding page in response. The client would then entirely replace the previous content with the new information sent by the server.

In the recent years, with the introduction of newer and richer technologies for web application development, web-applications have become more useable and interactive. These applications, called Rich Internet Applications (RIAs), changed the traditional web applications in two important aspects: first, client side-scripting languages such as JavaScript have allowed the modification of the web page by updating the Document Object Model (DOM) [16], which represents the client-side “state” of the application. Second, using technologies like AJAX [1] the client can communicate asynchronously with the server, without having the user to wait for the response from the server. In both cases, the URL typically does not change during these client side activities. Consequently, we can now have a quite complex web application addressed by a single URL.

These improvements increased the usability of web applications but on the other hand introduced new challenges. One of the important problems is the difficulty to automatically crawl these websites. Crawling is the process of browsing a web application in a methodical, automated manner or in an orderly fashion. Traditional crawling techniques are not sufficient for web applications built using RIA technologies. In traditional web application, a page is defined by its URL and all the pages reachable from the current page have their URL embedded in the current page. Crawling a traditional web application requires to extract these embedded URLs and traverse them in an effective sequence. But in RIAs, the current page can change its state dynamically sometimes without even requiring user input and hence cannot be mapped to a single URL. All these changes mean that traditional crawlers are unable to crawl RIAs, save for a few pages that have distinct URLs.

Crawling is an important aspect of the existence of the web. An important functionality of the web in general is the information it provides. This information can only be made available if the different information sources can be found and indexed. If search engines are not able to crawl websites with new information, they will not be able to index them. Hence a good part of the web in general will be lost. In addition, crawling is also required for any thorough analysis of the web application such as for security and accessibility testing. To our knowledge, none of the current search engines,web application testers and analyzers have the ability to crawl RIAs [2] .The problem gets increasingly important as more and more developers and organizations adopt these newer technologies to put their information on the web.

In this paper, we introduce a RIA crawling strategy using a statistical model. This strategy is based on the model-based crawling approach introduced in [3] to crawl RIAs efficiently. We evaluate the performances of our statistical model on two real RIAs and two test applications. We further compare our experimental results against other RIA crawling strategies, Depth-First, Breadth-First and Hypercube[3], and we show that the new strategy obtains overall better results.

The paper is organized as follows: in Section 2, we review related works. In Section 3, we give an overview of the model based crawling. In Section 4, we present the details of the new strategy based on statistical model. In Section 5, we provide experimental results obtained with our prototype, compared with existing crawling strategies. We conclude in Section 6 with some future research directions.

2 Related Works

For traditional web applications, crawling is a well-researched field with many efficient solutions in place [4]. However, in the case of RIAs the research is still ongoing to address the fundamental question of automatically discovering the existing pages. While at first glance web crawling may appear to be merely an application of

Breadth-First or Depth-First search, the truth is that there are many challenges in RIA crawling ranging from defining the states of the application and the state equivalence relation to efficiently discovering the information. Today's web applications have become much more complex and it might not be feasible to run the complete crawl. Thus, efficiency has been one prime factor that guides research directions for RIA crawling.

We survey here only research focusing on crawling RIAs. Although a limited topic, several papers have been published in the area of crawling RIAs, mostly focusing on AJAX applications. For example [5, 6, 7] focus on crawling for the purpose of indexing and search. In [8], the aim is to make RIAs accessible to search engines that are not AJAX-friendly. In [9] the focus is on regression testing of AJAX applications whereas [10] is concerned with security testing.

Mesbah et al. [11] introduced Crawljax, a tool for crawling AJAX applications that uses a variation of the Depth-First strategy. One of the drawbacks of the Crawljax strategy is that it uses its own logic to select only a subset of the possible "events" in the current state and thus might not be able to find all the states of the application. Moreover, Crawljax uses an edit distance (the number of operations that is needed to change one DOM to the other, the so-called Levenstein distance) to decide if the current state is different from the previous one. This approximation might incorrectly group some states together, leading to an incorrect model of the RIA being crawled.

Duda et al. [12] uses the Breadth-First strategy to crawl AJAX applications. They propose to reduce the communication costs of the crawler by caching the JavaScript function calls, together with the actual parameters.

Amalfitano et al. [13] focus on modeling and testing RIAs using execution traces. Their work is based on utilizing execution traces from user-sessions (a manual method). In a later paper [14], they introduced a tool, called CrawlRIA, which automatically generates execution traces using the Depth-First strategy.

All these papers suggest approaches to crawl RIAs, but to our knowledge there has not been much attention on the actual efficiency of crawling strategies. In particular, none of these techniques aims at discovering new states as early as possible in the crawl. Further, the Breadth-First and the Depth-First strategies are guaranteed to discover all states of the application when given enough time and under the right assumptions; however they are too generic and inflexible to be efficient when crawling most RIAs. There are opportunities to design more efficient strategies if we can identify some general patterns in the actual RIAs being crawled and use these patterns to come up with reasonable anticipations about the model of the application. We call this approach "model-based crawling". This concept was introduced in [3], in which we presented the crawling strategy based on Hypercube model. That strategy indeed presented an efficient way to crawl RIAs; however the assumptions made about the underlying model of the RIAs were too strict to be realistic for actual web applications.

3 Overview of Strategy

Crawling RIAs require crawler to understand more about a web page than just its HTML. It needs to be able to understand the structure of the document as well as the client side scripts such as JavasSript that manipulates it. To be able to investigate the deeper state of an application, the crawling process also needs to be able to recognize and execute "events" (which are occurences that cause client-side code execution and are typically triggered by user interactions such as clicking on an element) within the document to simulate the paths that might be taken by a real user. We call this an "event-based exploration". A Web-application can be conceptualized as a Finite State Machine with "states" representing the distinct DOMs that can be reached in the web application and transitions representing event executions . The result of crawling is called a "model" of the application. A model basically contains the states and the possible ways to move from one state to another.

3.1 Crawling Strategy

A crawling strategy is an algorithm that decides how the exploration proceeds. In the case of event-based exploration of RIAs, the strategy basically decides which event to explore next. We say that a crawling strategy that is able to find the states of the application early in the crawl is an efficient strategy, since finding the states is the goal of crawling. Efficiency is always important, but it is especially so in the case of RIA, because many RIAs are complex applications that have a very large state space. In such scenarios, it might not be feasible to wait for the crawler to complete the crawl. In this case, a strategy which discovers a larger portion of the application early on will deliver more data during the alloted time slot, and thus be more efficient. Generating more information about an application sooner not only helps search engines indexing more data but also allows security assessment and web application testing tools to provide more coverage as quickly as possible. However, the main problem is that we do not know how the web application has been built and without this prior knowledge of the web application, finding an efficient strategy is difficult.

One important aspect of the efficiency of the strategy is the use of "resets", that is, reloading the initial URL of the application to go back to the initial state. Resets are typically costlier (in terms of time of execution) than event execution. Moreover, resetting leads to the state that was reached initially when the crawl started, and most of the transitions from the initial state would typically have already been explored, so we have to go through known states and transitions before we could discover new states of the application.

Primarily motivated by the above goals, we introduced the concept of "model-based crawling" in [3] as:

1. First, a general hypothesis about the behaviour of the application is conceptualized. The idea is to assume that the application will behave in a certain way. Based on this hypothesis, one can define the anticipated (assumed) model of the application. This will transform the process of

crawling from the discovery acitivity to determine “what the model is” to the activity of validating whether the anticipated model is correct.

2. Once a hypothesis is elaborated and an anticipated model is defined, the next step is to define an efficient crawling strategy to verify the model. Without having an assumption about the behaviour of the application, it is impossible to define any strategy that will be efficient.
3. However, it is important to note that any real world application will never follow the anticipated model to its entirety. Therefore, we will also define a strategy which will reconcilates the differences discovered between the anticipated model and the real model of the application in an efficient way.

We define a two stage approach to confer to our primary goal of finding all states as soon as possible. The first phase is the “state exploration phase”. It aims at discovering all the states of the RIA being crawled. Once our strategy believes that it has probably found all the reachable states of the application, we proceed to the second phase, the “transition exploration phase” which tries to execute the remaining transitions after state exploration phase, to confirm that nothing has been overlooked. The motivation behind defining this two phase approach is that in many cases the application is too complex to be crawled completely, and it is important to explore, in the given time, as many states as possible, but unless we have explored all transitions in the application, we cannot be sure that we have found all states. In [3], we provided a solution where the underlying (anticipated) model was an hypercube. In this case, we were able to provide a strategy that was optimal for both phases.

3.2 Assumptions

An important factor for the efficiency of the crawling strategy is the definition of the state equivalence function. Equivalence function is used to determine whether a state can be regarded as being the same as another already seen. This is usually different from the simple state equality although, equality is one obvious equivalence function. The equivalence function plays a pivotal role in crawling of web-application. If an equivalence evaluation method is too stringent (like equality), then it may result in too many states being produced, essentially resulting in state explosion, long runs and in some cases infinite runs. On the contrary, if the equivalence relation is too lax, we may end up with client states that are merged together while, in reality, they are different, leading to an incomplete, simplified model. Our crawling strategy assumes that some valid equivalence function has been provided.

Another important factor is the deterministic behaviour of the web application. That is, from the same state executing the same event at different times will lead to the same state. The biggest challenge for making this assumption is the existence of server-side states, since during crawling we have access only to the client-side of the application. If there are server-side state changes then the application may behave differently when the same transition is taken at a different time from the same client-state since the server-state of the application may be different. Currently, we do not include server states in our crawling strategy (see [17] for more discussion on assumptions and requirements).

4 Probability Strategy

A crawling strategy can be efficient if it is able to predict the results of the event executions with some degree of accuracy. This helps prioritizing the events that are more likely to discover new states and hence improve efficiency. One way to have a strategy that can predict the outcomes of the events is to use a concrete anticipated model as we did in the Hypercube strategy [3]. But we can also make predictions about an event by looking at the outcomes of the previous executions of the same event from different states. Clearly, statistics about the past behavior of the event (from different states) can be used to model the future behavior of the event. With this motivation, we introduce a crawling strategy which makes use of the statistical data collected during crawling. The strategy is based on the belief that an event which was often observed to lead to new states in the past will be more likely to lead to new states in the future. We call this new strategy "the Probability strategy" as it estimates the probability of discovering a state for each event and prioritizes the events based on their probability.

In this section we provide an algorithm (crawling strategy) to select a state s and an event e in s for which our statistical model predicts the highest likelihood of discovering a new state. We start the discussion by giving the formula used to estimate events' probability of discovering new states.

4.1 Events' Probability of Discovering New States

Let P(e) be the event e's probability of discovering a new state. Remember that the same event "e" can be found in different states (we say that e is "enabled" in these states). The following Bayesian formula, known as the "Rule of Succession" in probability theory, is used to calculate P(e)

$$\mathrm{P(e)} = \frac{\mathrm{S(e)} + \mathrm{p_s}}{\mathrm{N(e)} + \mathrm{p_n}}$$

where:

- N(e) is the "execution count" of e, that is, the number of times e has been executed from different states so far.
- S(e) is the "success count" of e, that is, the number of times e discovered a new state out of its N(e) executions.
- p_s and p_n are the terms to represent initial success count and initial execution count respectively. These terms are preset and represent the initial probability of an unexecuted event to find a new state.

This Bayesian formula is useful for estimating the probabilities in situations when there are very few observations. Since at the beginning of the crawling we do not have any observations, this formula is suitable for our purpose. To use this formula we assign values to p_s and p_n to set the initial probability. For example, $p_s = 1$ and $p_n =$

2 can be used to set an event’s initial probability to 0.5 (note that N(e) = S(e) = 0 initially).

Having Bayesian probability instead of using the “classical” probability, P(e)=S(e)/ N(e), with some initial values for P(e), avoids in particular have events that get a probability of 0 because no new state were found at their first execution. With our formula, events never have a probability of 0 (or 1) and can always be picked up after a while.

After each execution of an event from a state where the event was not executed before, the event’s probability will be updated by taking into account the result of this recent execution. As we execute an event and have more observations about its behaviour, the weight of the initial probability will decrease and actual observations will dominate the value of the probability.

4.2 Choosing Next Event to Explore

In this section, we describe the logic that helps the strategy decide which event to explore next. Before going into details, we introduce the notation and definitions used.

- S denotes the set of already discovered states. Initially S contains the initial state and each newly discovered state is added to S as the crawl progresses.
- $s_{current}$, represents the current state, the state we are at currently in the application while executing the crawl. $s_{current}$ always refers to one of the states in S.
- For a state s, we define the probability of the state, P(s), as the maximum probability of an unexecuted event in s. An unexecuted event of s is an event that is enabled in s but has not yet been explored by the strategy from s. If s has no unexecuted events than P(s) = 0
- d(s, s') is the distance from $s \in S$ to $s' \in S$. It is the length of the shortest path from s to s' in the known model of the application.

When deciding which event to explore next, the Probability strategy aims to take the action that will maximize the chances of discovering a new state while minimizing the cost (number of event executions and resets). For this reason, starting from the current state $s_{current}$, we search for a state s_{chosen} such that executing the event with probability $P(s_{chosen})$ in s_{chosen} achieves this goal.

All the states that still have unexecuted events are the candidates to become the chosen state. However we have to take into account the distance from the current state to the chosen state in addition to the raw probabilities. Note that from $s_{current}$ reaching to any other state in S means following a known path (consisting of already executed events). By traversing a known path, we will not be able to discover new states. Between two states that are at different distances from $s_{current}$, we may consider reaching the one that is farther away because of its higher probability. However, it requires traversing a longer known path. The time to execute the extra events in this path could actually be used for exploration if the closer state is chosen. As will be explained shortly, to make decisions in such situations we need to balance the cost of executing known event with the higher probability of the farther state.

For our analysis it is necessary to have an estimation of discovering a state by executing an (unexecuted) event from an "average" state in S. For this purpose, we will use the average probability P_{avg} that is defined as follows.

$$P_{avg} = (\Sigma_{s\in S} P(s)) / |S|$$

To select a state that maximizes the probability while minimizing the cost, we need a mechanism that compares two states and decides which is more preferable. Let's say we want to compare two states s and s'. If the two states are at the same distance from the current state (i.e. $d(s_{current}, s) = d(s_{current}, s')$) then the one with the higher probability is obviously a better choice. But if the cost of reaching one of the states, is higher than the other, say $d(s_{current}, s) < d(s_{current}, s')$) then there can be two cases. If $P(s) \geq P(s')$ then s is clearly a better choice. But if $P(s) < P(s')$ then the fact that reaching s' is costlier than reaching s should be reflected in the comparison mechanism. To make up for the difference in the cost, we should allow the exploration of a sequence $k = d(s_{current}, s') - d(s_{current}, s)$ extra events after executing the event with probability P(s) from s. Thus we use the probability of discovering a new state after executing the event from S and executing k more unexecuted events (each with a probability of P_{avg} to discover new state). This is given by the following formula

$$1 - \big(1 - P(s)\big)\big(1 - P_{avg}\big)^{d(s_{current},\ s) - d(s_{current},\ s')} \qquad (1)$$

Now we can compare this value with P(s') and choose the option that gives the higher probability.

In the following, we will present an algorithm that chooses from the set of discovered states, S, a state, s_{chosen}, such that executing the event with maximum probability on s_{chosen} maximizes the probability of discovering a state while minimizing the number of event executions. In particular, the s_{chosen} that we are looking for is the state, $s \in S$ that satisfies the following condition

$\forall\ s' \in S$

- if $d(s_{current},\ s) = d(s_{current},\ s'),\ P(s') \leq P(s)$
- if $d(s_{current},\ s) < d(s_{current},\ s'), 1 - \big(1 - P(s)\big)\big(1 - P_{avg}\big)^{d(s_{current},\ s') - d(s_{current},\ s)} \geq P(s')$
- if $d(s_{current},\ s) > d(s_{current},\ s'),\ 1 - \big(1 - P(s')\big)\big(1 - P_{avg}\big)^{d(s_{current},\ s) - d(s_{current},\ s')} < P(s)$

The first case makes sure that s is a better choice than any other state, s' which is at the same distance from the current state as s. The second case makes sure that s is a better choice than any state, s' which is farther from the current state than s is. The last case makes sure that s is a better choice than any state, s' which is closer to current state than s is.

4.3 The Algorithm

In this section we give an algorithm that can efficiently decide on a s_{chosen} that satisfies the condition given in the previous section. The algorithm goes over the states in S in a systematic fashion to find the state that satisfies the condition to be s_{chosen}. The algorithm initializes the variable s_{chosen} to the current state and searches for the s_{chosen} in iterations. At iteration i the states at a distance i from the current state are compared against the state currently referenced by s_{chosen}. Using the given condition in previous section, we check if there is any state at distance i from the current state more preferable to $s_{chosen.}$

```
Algorithm ChooseStateToExplore
$s_{chosen}$ := $s_{current}$ ;
i := 1;
distanceToCheck = maxDistanceToCheckFrom($s_{chosen}$);
while ( i < distanceToCheck) {
      for each s where d($s_{current}$, s) = i {
            if (s is preferable to $s_{chosen}$) {
                  $s_{chosen}$ := s;
                  distanceToCheck += maxDistanceToCheckFrom($s_{chosen}$);
            }
   }
   i++;
}
return $s_{chosen}$;
```

We can optimize the search by exploiting the fact that we do not necessarily need to explore all the states in S to find s_{chosen} .The search can be stopped the moment we detect that it is not possible to find any state further away with a higher probability. This is possible since we take into account the cost of distance while comparing the probability of states. We only need to know the probability of the state with maximum probability in S. We call this probability P_{best}.

Then the maximum distance that needs to be considered from s_{chosen} (this distance is noted in the algorithm as maxDistanceToCheckFrom(s_{chosen})) is the value of smallest d that satisfies

$$1 – (1- P(s_{chosen}))(1 - P_{avg})^{d}) \geq P_{best} \qquad (2)$$

When the left hand side of (2) becomes as large as P_{best} then it is not required to look further since even the states that might have the maximum probability, P_{best}, will not be preferable anymore to s_{chosen} due to the distance factor.

5 Experimental Results

In this section, we evaluate the performance of the Probability strategy on two real RIAs and two test RIAs. We have used the following metrics for performance evaluation.

(1) Number of events and resets required to discover all states

(2) Number of events and resets required to explore all transitions

However, for simplicity we have combined the events and resets required for state exploration and transition exploration into a single cost factor. For this purpose, we have simply expressed the cost of resets in terms of number of event execution (the actual value used is application dependent and is based on experiments). We believe that number of events execution is a good metric for performance evaluation, since the time to crawl is proportional to the number of events executed during the crawl.

We compare the performance of our model with the Breadth-First and the Depth-First strategies. We compare with these strategies as these are standard graph exploration strategies and most of the published crawling results have used a variation of these standard strategies [5, 11, 14]. It is important to mention that our implementations of the Breadth-First and the Depth First strategies are optimized to use the shortest known path to reach the next state to explore (as opposed to using systematically resets, which is very inefficient).

We also present, for each application the optimal number of events and resets required to explore all the states of the application. It is important to understand that this optimal value is calculated after the fact, once the model of the application is obtained. In our case the optimal path to visit every state of the application can be found by solving the Asymmetric Traveling Salesman Problem (ATSP) on the graph instance obtained for the application. In Traveling Salesman Problem, given a list of cities and their pairwise distances, the task is to find the shortest possible route that visits each city exactly once. The ATSP differs from its symmetric counterpart in the sense that paths may not exist in both directions or the distances might be different, forming a directed graph. This seems a reasonable strategy as a web application can be modeled as directed graphs with states as nodes and event executions as directed edges. Before calculating the optimal cost, we define the pair wise distance or cost between all pairs of states. This is possible as all states are reachable from the home state and from every state we can reach the home state, possibly using reset.

We have used an exact ATSP solver [15] to get the optimal path. From this path, optimal number of event executions and resets to discover all states is obtained.

In an effort to minimize any influence that may be caused by considering events in a specific order, the events at each state are randomly ordered for each crawl. Also, each application is crawled 5 times with each method and the average cost of these five runs is used for comparison. Further, for all strategies and applications the cost of reset is calculated separately. The value is chosen by comparing the time it takes to load the initial page and average execution time of an event in our tool.

5.1 Test Applications

The first real RIA we consider is an AJAX-based periodic table[1]. The periodic table contains the 118 chemical elements in an HTML table. Clicking on each cell containing a chemical element retrieves detailed information about the element. In total 240 states and 29034 transitions are identified by our crawler and the reset cost is 2.

The second real application considered is Clipmarks[2]. Clipmarks is a RIA which allows its users to share parts of any webpage (images, text, videos) with other users. For this experimental study we have used a partial local copy of the website in order to be consistent for the each strategy. The partial local copy of the website consisted of 129 states and 10580 transitions and the reset cost is 18.

The third application, TestRIA[3], is a test application that we developed using AJAX. TestRIA mimics standard company or personal webpage. Each state contains menu items such as home, contact us etc. TestRIA has 39 states and 105 transitions and a reset cost of 2.

The fourth application[4] is a demo web application maintained by the IBM® AppScan® Team. We have used the AJAX-fied version of the website. The application has 45 states and 305 transitions and a reset cost of 2.

5.2 State Exploration Phase

For compactness we have used boxplots: the top of vertical lines show the maximum number required to discover all the states.The lower edge of the box, the line in the box and the higher edge of the box indicate the number required to discover a quarter, half and 3 quarters of all the states in the application, respectively. We will use the position of this box and the middle line to assess whether a method is able to discover new states faster than others. The boxplots are drawn in logarithmic scale.

It is important to mention that we are only interested in two factors to define the efficiency of the crawling algorithms. First, the cost required to discover all the states of the application. Once we have discovered all the states of the application the only important factor that remains is the total cost to perform a complete crawl of the reaming transitions. The cost of the state exploration phase is important as it might not be feasible to finish the crawling and we would want to explore as much states as possible within the given run of the algorithm. Hence it is very important to find what percentage of the total state space have been discovered by the crawling algorithm at a given time during the crawl. However, once all the states have been discovered the only factor remaing is the cost that the crawling algorithm takes to finish all the remaining transitions.

[1] http://code.jalenack.com/periodic (Local version: http://ssrg.eecs.uottawa.ca/periodic/)

[2] http://www.clipmarks.com/ (Local version: http://ssrg.eecs.uottawa.ca/clipmarks/)

[3] http://ssrg.eecs.uottawa.ca/TestRIA/

[4] http://www.altoromutual.com/

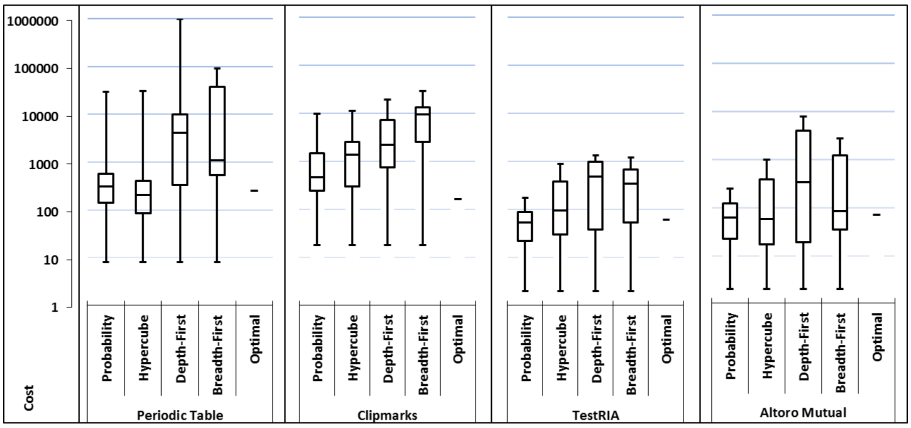

Figure1: State exploration costs (Logarithmic scale)

The plots above make a very convincing statement about the efficiency of the Probability strategy. The paper [3] proved the efficiency of the Hypercube strategy compared to the current state-of-the-art commercial products and other research tools. It is thus useful to provide a comparative analysis of our strategy against the Hypercube strategy.

For the first real RIA website (periodic table), the Probability strategy performs significantly better than the Breadth-First and the Depth-First both in terms of the final cost and the progress during the crawl. The Depth-First and Breadth-First strategies are both very expensive with total costs of 971701 and 51104 events execution respectively. The Probability strategy also performs very good compared to the Hypercube strategy. The progress of the crawl of both the algorithms is very competitive, while the Probability strategy has a smaller total cost of 28393 event executions compared to 29582 for hypercube. Further, the Probability also performs really well in terms of the optimal cost of the crawls only differing significantly towards the last quarter of the crawl.

We see very similar results for the second real RIA (clipmarks). Again the Probability strategy performs better in terms of a total cost of 8235 event executions, the least among all the crawling strategies. In addition, it surpasses the Hypercube strategy's performance during the progress of the craw, where the Hypercube results in a total cost of 8688 event executions. As expected, the Probability strategy performs better than both the Depth-First and the Breadth-First which take 11980 and 15393 event executions respectively.

On the test application (TestRIA), the Probability strategy performs very close to the optimal solution and outperforms all other strategies by a factor of more than 5 with a total cost of 86 event executions. This is exceptionally better than the other strategies that have values of 493, 329 and 528 for the Hypercube, the Depth-First and the Breadth-First respectively.

The last application (Altoro Mutual) is a demo website developed to model the typical operation of a bank website. As we can see, the Probability strategy again performs very close to the optimal solution with a total cost of 121 event executions.

The progress of the crawl of the Probability strategy is very similar to the optimal solution. In addition, it outperforms the other strategies by a factor of 10 or more, having a total cost of 603, 3713 and 1418 for the Hypercube, the Depth-First and the Breadth-First respectively.

5.3 Transition Exploration Phase.

As explained above, we are only interested in the overall cost of the crawl in terms of event executions (the cost of exploring all transitions). As we can see, the Probability strategy is again the best in the total overall cost of crawling. For most of the websites the Probability strategy's performance is better than or comparable to the Hypercube strategy but it significantly outperforms the Depth-First and the Breadth-First strategies.

Table1: Transition Exploration Costs

	Periodic Table	Clipmarks	TestRIA	Altoro Mutual
Probability	31403	9717	979	2526
Hypercube	31810	11233	996	2542
Depth-First	983582	19128	1345	7693
Breadth-First	181924	28710	1324	3744

When compared to the Hypercube strategy, the Probability strategy is simple to understand and implement. Hypercube strategy requires strict assumptions about the web application and involves complex algorithms that will probably not be understood by most. So our conclusion is that the Probability model is a better choice, much simpler and actually slightly more efficient than the Hypercube.

When compared to the optimal solution, we see that there is still some room for improvement, but we are closing in. We state again that the optimal solution is calculated after the website model is known, and thus can only be used as a benchmark, not to actually crawl an unknown web application. The most striking factor about the Probability strategy is that it was never the worst performer and best in most of the cases. Further, its performance was significant both with respect to general purpose website and also on websites developed to model specific requirements.

6 Conclusion and Future Work

In this paper, we have presented a new crawling strategy based on the idea of model based crawling introduced in [3]. Our strategy aims at finding most of the states of the application as soon as possible but still eventually finds all the states and transitions of the web application. Experimental results show that this new strategy performs very well and outperforms the standard crawling strategies by a significant margin. Further, it also outperforms the Hypercube strategy [3] in most cases and it performs comparably in the least favorable example, while being very much simpler to understand and to implement. However, there is a lot more to be explored in the area of RIA crawling.

First, we are trying to expand the concept of state equivalence to form a notion of independent and dependent states and use it to crawl websites like Google calendar, where most of the states are independent of each other and can be crawled individually.

Second, we are exploring other models apart from the Probability and the Hypercube. Probability model is a very general model and doesn't exploit the structure of the web application completely to its advantage. On the other hand Hypercube makes very strict assumption about the structure of the web application. We are trying to explore notions that can gain advantage from the structure of the application but does not make very strict assumptions.

A third direction would be to enhance the model with the notion of "important" states and events, that is, some states would have priority over others and some events would be more important to execute than others. We believe that the model-based crawling strategy can be adjusted to deal with this situation.

Finally, a fourth direction is exploring and using the model based crawling strategy for dictributed crawling. We are working on distributed crawling algorithms and exploiting the cloud services for the purpose of web application crawling.

Acknowledgments: This work is supported in part by IBM and the Natural Science and Engineering Research Council of Canada.

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

References

1. Jesse James Garrett. (2005) Adaptive Path. [Online]. "http://www.adaptivepath.com/publications/essays/archives/000385.php"
2. Bau, J., Bursztein, E., Gupta, D., and Mitchell, J.C.: State of the Art: Automated Black-Box Web Application Vulnerability Testing. In Proc. IEEE Symposium on Security and Privacy. (2010).
3. Benjamin, K., Bochmann, G., Dinturk, M.E., Jourdan, G.-V. AND Onut, I.V., 2011. A Strategy for Efficient Crawling of Rich Internet Applications. In S. Auer, O. Díaz & G. Papadopoulos, eds. Web Engineering: 11th International Conference, ICWE 2011, Paphos, Cyprus. Springer Berlin / Heidelberg. 74-89.
4. Olston, C. and Najork, M., 2010. Web Crawling. Foundations and Trends in Information Retrieval, 4(3), 175-246. Available at http://dx.doi.org/10.1561/1500000017.
5. Duda, C., Frey, G., Kossmann D., and Zhou, C.: AJAXSearch: Crawling, Indexing and Searching Web 2.0 Applications.VLDB. (2008).
6. Frey, G.: Indexing Ajax Web Applications, Master's Thesis, ETH Zurich. (2007).

7. Matter, R.: Ajax Crawl: making Ajax applications searchable. Master's Thesis. ETH, Zurich. (2008).
8. Mesbah, A., and Deursen, A. v.: Exposing the Hidden Web Induced by AJAX. TUD-SERG Technical Report Series. TUD-SERG-2008-001. (2008).
9. Roest, D., Mesbah, A., Deursen, A. v.: Regression Testing Ajax Applications: Coping with Dynamism. Third International Conference on Software Testing, Verification and Validation. pp.127-136. (2010).
10. Bezemer, B., Mesbah, A., and Deursen, A. v.: Automated Security Testing of Web Widget Interactions. Foundations of Software Engineering Symposium (FSE). ACM. pp. 81–90. (2009).
11. Mesbah,A., Bozdag, E., and Deursen, A.v.: Crawling Ajax by Inferring User Interface State Changes. In Proceedings of the 8th International Conference on Web Engineering, IEEE Computer Society, 122-134. (2008).
12. Duda, C., Frey, G., Kossmann, D., Matter, R. AND Chong Zhou, 2009. AJAX Crawl: Making AJAX Applications Searchable. In IEEE 25th International Conference on Data Engineering., 2009. 78-89.
13. Amalfitano, D., Fasolino, A. and Tramontana, P., 2008. Reverse Engineering Finite State Machines from Rich Internet Applications. In Proc. of 15th Working Conference on Reverse Engineering. Washington, DC, USA, 2008. IEEE Computer Society. 69 -73.
14. Amalfitano, D., Fasolino, R. and Tramontana, P., 2010. Rich Internet Application Testing Using Execution Trace Data. In Proceddings of Third International Conference on Software Testing, Verification, and Validation Workshops. Washington, DC, USA, 2010. IEEE Computer Society. 274-283.
15. Carpento, G., Dell'amico, M. and Toth, P., 1995. Exact solution of large-scale, asymmetric traveling salesman problems. ACM Trans. Math. Softw., 21(4).
16. World Wide Web Consortium (W3C). (2005) Document Object Model (DOM). [Online]. http://www.w3.org/DOM/
17. Benjamin, K., Bochmann, G.v., Jourdan, G.V., and Onut, I.V.: Some Modeling Challenges when Testing Rich Internet Applications for Security. In First International workshop on modeling and detection of vulnerabilities. Paris, France. (2010).