Model-based Crawling - An Approach to Design Efficient Crawling Strategies for Rich Internet Applications

Mustafa Emre Dincturk

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for a doctoral degree in Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Rich Internet Applications (RIAs) are a new generation of web applications that break away from the concepts on which traditional web applications are based. RIAs are more interactive and responsive than traditional web applications since RIAs allow client-side scripting (such as JavaScript) and asynchronous communication with the server (using AJAX). Although these are improvements in terms of user-friendliness, there is a big impact on our ability to automatically explore (crawl) these applications. Traditional crawling algorithms are not sufficient for crawling RIAs. We should be able to crawl RIAs in order to be able to search their content and build their models for various purposes such as reverse-engineering, detecting security vulnerabilities, assessing usability, and applying model-based testing techniques. One important problem is designing efficient crawling strategies for RIAs. It seems possible to design crawling strategies more efficient than the standard crawling strategies, the Breadth-First and the Depth-First. In this thesis, we explore the possibilities of designing efficient crawling strategies. We use a general approach that we called *Model-based Crawling* and present two crawling strategies that are designed using this approach. We show by experimental results that model-based crawling strategies are more efficient than the standard strategies.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Guy-Vincent Jourdan, for his invaluable guidance and support.

I cannot be thankful enough to Dr. Gregor von Bochmann and to Dr. Iosif Viorel Onut for their advice and support throughout my studies.

I would like to thank the members of the thesis committee, Dr. Nejib Zaguia, Dr. Babak Esfandiari, Dr. Liam Peyton and Dr. James Miller for their constructive feedback and their investment of time.

I am thankful to my MSc. supervisor, Dr. Hüsnü Yenigün, for encouraging me to pursue a Ph.D.

Many thanks to Seyed M. Mirtaheri, Ava Ahadipour, Ali Moosavi, Kamara Benjamin, Suryakant Choudhary, Khaled Ben Hafaiedh, Bo Wan and Di Zou for accompanying me in this journey.

I would like to acknowledge the financial support of the IBM Center for Advanced Studies (CAS) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

My special thanks to my family and to Gülden Sarıcalı for their love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A Web application is an application that is accessed over the Web (usually using HTTP over TCP/IP) and provides content and services using HTML documents. Web applications follow a client-server architecture. Web browsers (the client-side) provide convenient and platform-independent access to the information stored in the server. In the early years of the Web, all the processing was done on the server-side and the client-side was used only as a user interface. Over the last two decades, new web technologies changed this situation by gradually adding more processing capability on the client-side and employing more flexible communication patterns between the client and the server. This resulted in more responsive and interactive web applications, so-called Rich Internet Applications (RIAs). However, existing techniques for automatic exploration (crawling) of web applications became insufficient for RIAs.

This thesis discusses the impact of the RIA technologies on crawling and addresses the problem of designing efficient crawling strategies for RIAs. In this chapter, we look at the differences between traditional web applications and RIAs, introduce the basic concepts, and explain our motivation for this research.

1.1 Traditional Web Applications

A traditional web application is a collection of static web pages generated exclusively on the server-side. Web pages are hypertext documents encoded in Hypertext Markup Language (HTML). In addition to the content (such as text, images and tables), HTML documents may contain references, called hyperlinks, to the other resources on the Web. A hyperlink references a resource using a Uniform Resource Locator (URL). A URL

contains the information required to access the resource: the name of the resource, the host (server) address where the resource is located and the name of the protocol to use to access the resource.

HTTP (Hypertext Transfer Protocol) is the communication protocol between the web client and the server. HTTP is an application layer protocol using a request-response style of communication. The basic request methods of HTTP are GET and POST, which are used to retrieve content from the server. A POST request also allows to submit an arbitrary amount of user data to the server. Using these methods, the client requests a resource by providing the URL of the resource. When the server receives the request, it locates the resource and sends it as a response.

A Web browser is a software that is used on the client-side to view and navigate web pages. In a browser, when the user enters a URL of an HTML document, the browser retrieves the document from the server and renders a visual representation of the document. By clicking on a hyperlink found on the currently viewed HTML document (or entering a URL directly), the user causes the browser to generate a new request. When the response arrives, the browser loads the new page replacing the old one completely. Traditional web applications use this synchronous communication pattern where the user interaction is blocked until the response arrives and replaces the current page.

In the first years of the Web, the web pages were static. That is, they were prepared and stored in the server in advance. Later, server-side technologies (such as CGI, PHP, JSP and ASP) allowed pages to be tailored for each client and each request. When a page is requested from the server, instead of returning a static HTML file stored in the file system, the server could dynamically generate an HTML file based on the user-specific data such as the parameters in the URL, the values entered in an HTML form, or the type of the web client used. But, these applications are still considered "traditional applications" since they consist of some URL-addressable web pages that are retrieved synchronously from the server.

1.2 Rich Internet Applications

Rich Internet Applications (RIAs) are a newer generation of web applications that are more interactive and responsive than the traditional ones. RIAs achieve this with two essential enhancements: First, RIAs add flexibility to the client-side by allowing the browser to carry out computation related to the application logic. That is, the server can send, along with the HTML page, the scripts (such as JavaScript) that can be executed

client-side when a user interaction takes place. These scripts are capable of modifying the web page without contacting the server. Second, RIAs introduce an asynchronous communication pattern. That is, these scripts can generate and send new requests to the server while the user continues interacting with the application without waiting for the responses of these requests.

There are several RIA technologies such as AJAX (Asynchronous JavaScript and XML) [40], Flex [11], Silverlight [56] and Java Applets that can provide the browser these enhancements. Except AJAX, all these technologies require a plug-in to be installed on the browser. AJAX uses established web standards (JavaScript and XML), which are supported by all major browsers without additional plug-ins. In this thesis, we focus on AJAX applications but the concepts are applicable to the other similar technologies as well.

1.2.1 JavaScript and Document Object Model

JavaScript is a programming language that is primarily used to add code to the client-side of a web application. Browsers are capable of interpreting and running JavaScript code. Browsers also implement a platform and language independent interface, called DOM (Document Object Model) [66] that allows computer programs to access and modify the contents, style and structure of HTML and XML documents.

A *DOM instance* is the internal (in-memory) representation of an HTML document in the browser. To render an HTML document, the browser first parses the document and creates the DOM instance by creating the objects (DOM elements) for the HTML elements and their attributes. These objects are bound together in a tree structure (DOM tree) representing the hierarchical relationships of the HTML elements in the document. The DOM instance is then used to create a visual rendering of the HTML content.

Using the DOM interface methods, a JavaScript code is able to access and manipulate the DOM instance that is currently shown to the user. When some JavaScript code modifies the DOM instance, the browser reflects these modifications to the user by re-rendering the modified parts.

DOM Events

In a browser, JavaScript execution is triggered by an event.

> *Event*: An *event* is an action (or time-out) that causes JavaScript code execution.

Every event is associated with a DOM element. The DOM interface specifies the possible types of events for DOM elements. Two example events are onclick (the user clicks on an element) and onload (an element, usually a page, frame, or image, has just been loaded). When an event occurs, the browser detects the element on which the event happened and executes all event handlers that are registered to the element for that event.

> *Event Handler*: An *event handler* is some JavaScript code that runs as a reaction to an event.

1.2.2 AJAX

AJAX (Asynchronous JavaScript and XML) [40] is a technique that is used for communicating with the server asynchronously. Asynchronous communication for web applications means that when a request is made to the server, the browser does not block user interactions while waiting for the response to come. Instead, the browser allows the user to continue interacting with the page and possibly generate new requests. This is in contrast to the synchronous communication pattern in traditional web applications. Figure 1.1 and Figure 1.2 show the two communication patterns.

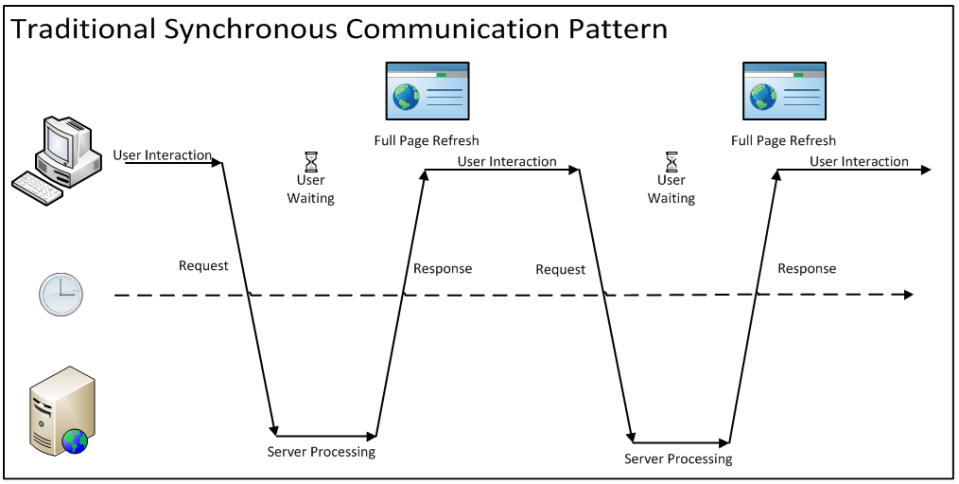

Figure 1.1: Synchronous Communication in Traditional Applications

In AJAX, an asynchronous request is achieved using an instance of the JavaScript object called XMLHttpRequest. This object is capable of sending the standard HTTP requests (usually GET and POST). When using this object, the JavaScript code that must be run to handle the response data, called the callback method, should be specified.

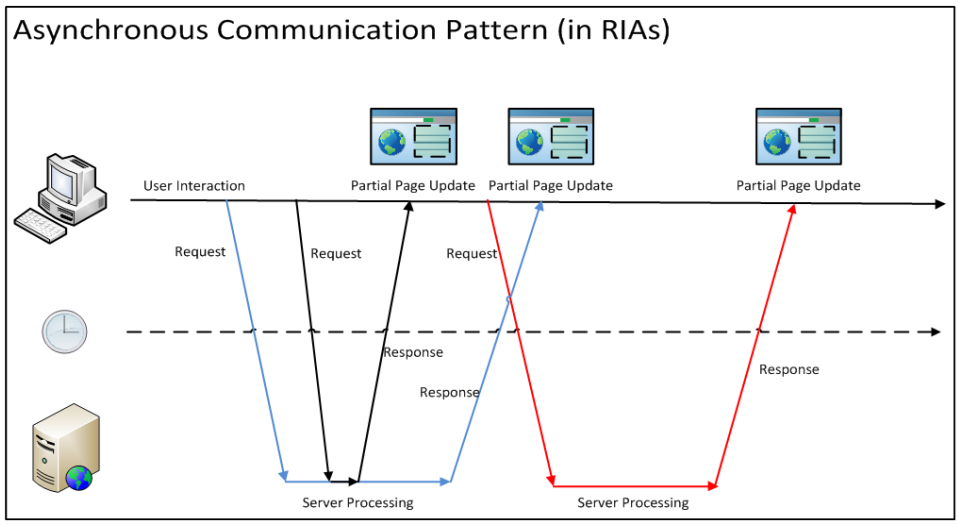

Figure 1.2: Asynchronous Communication in RIAs

When the response arrives, the browser runs the specified callback method which may modify the DOM instance using the data received. The AJAX technique makes continuous user interaction and partial page updates possible. AJAX eliminates the requirement for complete page refresh each time a request is made. Instead, the information received from the server may now be used to change a portion of the web page. While some parts of the page is being changed, the user can still interact with the other parts.

1.3 Crawling Web Applications

Crawling is the process of exploring a web site or an application automatically. A crawler is a tool that performs crawling. The crawler aims at discovering the web pages of a web application by navigating through the application. The crawler behaves like any other user in the sense that it simulates possible user interactions and can only observe the client-side of the application. However, the crawler explores the application in an automated and more efficient manner.

1.3.1 Motivations for Crawling

There are several important motivations for crawling, for example, content indexing, automated testing, and performing automated analyses such as security vulnerability detection and usability assessment.

To search for information on the Web, Web users rely on search engines. In order for search engines to know what information is available on the Web, they have to keep looking for new pages and index their content. Without crawling and indexing, the information in these pages would be very hard to find for Web users.

Since web applications are also used for providing sensitive data and services, there are often concerns with the security and the usability of these applications. To address these concerns, many commercial and open-source automated web application scanners exist [15, 32]. These tools aim at detecting possible issues in an application, such as security vulnerabilities and usability issues, in an automated and efficient manner. These tools have to use crawlers to discover the existing pages in the application so that they can analyze the pages for the targeted issues.

1.3.2 Model of an Application

The result of crawling is called a *model* of the application. A model of the application essentially contains the existing pages (DOM instances) and the ways to move from one page to another. A model can be thought as a State Machine consisting of states and transitions:

> *State*: A *client-state* (or simply state) is the state of the application as it is seen on the client-side. States represent DOM instances.
>
> *Transition*: A *transition* represents the execution of an event that leads the application from one state to another.

The crawler builds a model of the application starting from a given page (or pages) of the application (normally the home/index page), simulating the user actions on the DOM instances it encounters, augmenting the current model by adding new states for each new DOM instance and recording the events that cause change from one state to another as transitions.

1.3.3 Requirements

Below, we list some requirements that we believe are important for a crawler to satisfy when building a model of the application.

- **Correctness of the Model:** The extracted model should be a correct representation of the explored parts of the application.

- **Deterministic Crawl:** The crawling algorithm should be deterministic. That is, when the same unchanged application is crawled twice, the same model should be produced by the crawler. This is important since we want to be able to use the model produced by the crawler for analyzing the discovered pages. The model should be reproducible as long as the web application is not changed.

- **Completeness of the Model:** We believe that the crawler should be able to build a complete model of the application when given enough time. That is, when we remove any time constraint on the crawler, it should guarantee that any reachable state in the application will eventually be found. Also, the crawler should be able to capture the necessary information to reach a discovered state. This information should include the variables and user-inputs that may be necessary to trigger a transition, in addition to the links or the events. Completeness is important since we want all the pages of the application to be available for analysis or indexing at some point during the crawl.

- **Efficiency of the Crawl:** The model should be built in an efficient manner. For us, being efficient means that the crawler should discover as much of the model as quickly as possible. Since the valuable information is present in the states rather than the transitions, an efficient crawler aims at discovering the states of the application first rather than exploring the transitions between already known states. This is important since the crawl may not terminate in a feasible amount of time for some large applications,. In that case, even if it is not possible to wait the crawl to finish, we desire crawler to discover the maximum amount of information in the time it is allowed to run. Thus, we can index or analyze more content from the application.

We note that the last two requirements are not universal requirements. Some may argue that the completeness is not required since one may only be interested in the pages that seem "important" rather than exploring everything, or some may define the crawling efficiency in a different way by saying crawling efficiency is about deciding what parts of the application to crawl. For example, it can be argued that exploring a state that is more than 3 clicks away from the initial page is not necessary and leads to inefficiency since most real users do not go that deep in the application. For us, the concept that is referred to with such arguments is the effectiveness of the crawl rather than efficiency. Effectiveness of the crawl can be defined as discovering important pages first given an

importance metric. Even if the aim is to just to discover a subset of the existing pages (the important ones), it is still important to discover all these pages in an efficient manner.

1.3.4 Crawling Traditional Web Applications

Crawling traditional applications is relatively easy compared with crawling Rich Internet Applications. In a traditional application, each page is addressed by a URL. Hence, the states of a traditional application can be identified based on the URLs. The basic task for the crawler is to find the URLs in the application. The algorithm to achieve this is simple. The crawler starts with a given set of URLs (seeds) of the application and removes a URL from this set, downloads the corresponding page, from the page extracts all the URLs that are not seen before and adds them to the set. The process is repeated until the set becomes empty. We refer to this type of exploration as "*URL-based crawling*".

1.3.5 Crawling Rich Internet Applications

URL-based crawling is not sufficient for crawling RIAs since the assumption that each state is addressed by a URL is not valid in RIAs. In RIAs, client scripts can change the state without changing the URL. It is not uncommon for a RIA to have a single URL, where the whole application is navigated via events.

In order to crawl RIAs, it is necessary to execute the events found in each state. To achieve this, the crawler should analyze each DOM instance and identify the DOM elements that have registered event handlers. Then, the crawler can execute these event handlers as if a user interaction took place and see how the DOM instance changes. We refer to the execution of an event handler by the crawler simply as an *event execution*.

When crawling a RIA, our goal is to start from a state that can be directly reached by a URL and extract a model that contains all the states reachable by event executions. We refer to this type of exploration as *event-based crawling*.

Model Representation

The model that is built for a URL by event-based crawling can be conceptualized as a Finite State Machine (FSM). We formulate an FSM as a tuple $M = (S, s_1, \Sigma, \delta)$ where

- S is the *finite set of states*
- $s_1 \in S$ is the *initial state* of the URL

- Σ is the set of all events in the application
- $\delta : S \times \Sigma \rightarrow S$ is the *transition function*

The *initial state* s_1 is the state that represents the DOM instance reached when the URL is loaded.

During exploration, the application can be in only one of its states, referred to as the *current state*.

For two states s_i and s_j and an event e if $\delta(s_i, e) = s_j$ then the application (modeled by the FSM M) performs a transition from the state s_i to the state s_j when the event e is executed in s_i. $(s_i, s_j; e)$ denotes such a transition. The state from which the transition originates (s_i) is called the *source* state and the state to which the transition leads (s_j) is called the *destination* state of the transition.

The transition function, δ, is a partial function: it may be that only a subset of the events in Σ can be executed from a given state s. This subset contains the events associated with elements existing in the DOM represented by s. It is called the *enabled events* at s.

M is deterministic under our working assumptions which are explained in Chapter 2.

An FSM $M = (S, s_1, \Sigma, \delta)$ can be represented as a directed graph $G = (V, E)$ where

- V is the set of vertices such that a vertex $v_i \in V$ represents the state $s_i \in S$
- E is the set of labeled directed edges such that $(v_i, v_j; e) \in E$ iff $\delta(s_i, e) = s_j$. When it is not important, we omit the edge's event label and simply write (v_i, v_j).

In a graph, any sequence of adjacent edges is called a path. We note the concatenation of two paths P and P' by juxtaposition: PP'. Given paths P, P', P_P, P_S, we say P' is a subpath of P if $P = P_P P' P_S$, where P_P and P_S are (possibly empty) prefix and suffix of P, respectively. The length of a path P is the number of edges in P.

Building a Complete Model

Under our working assumptions (explained in Chapter 2), we aim at building a complete model. Building a complete model means that every state (and every transition) will eventually be discovered. Achieving this requires to execute each enabled event at each discovered state.

The same event can be enabled in multiple states. It is not enough to execute the event only from one of the states since executing the same event from a different state

may lead to a different state. For example, Figure 1.3 shows a simple model for a RIA where the content of the application can be navigated using the "next" and "previous" events. Notice that each instance of the event "next" (or "previous") leads to a different state, depending on where it is executed. To make sure that no state is missed, the crawler should execute from each state all the enabled events at least once.

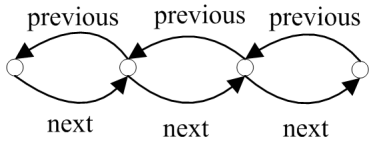

Figure 1.3: Model of a simple RIA

In the remainder of the thesis, we refer to the first execution of an event e from a state s as the "*exploration* of e from s" (sometimes, we say a transition is explored to mean that the corresponding event is explored from the source state).

When building a model for a given URL, initially we start with a single vertex representing the initial state reached by loading the URL. Then, the crawler identifies the enabled events on the initial state and explores one of the enabled events. After each event exploration, the model is augmented by adding a new edge for the newly discovered transition. If a new state is discovered, a new vertex is added to the model. When each enabled event at each discovered state is explored, a complete model is obtained for the URL.

A complete crawling strategy for RIAs should apply event-based crawling in addition to URL-based crawling since there are RIAs that use the traditional URL-based navigation together with the event-based navigation. In this case, for each discovered URL, the event-based crawling needs to be done if there are events enabled on the corresponding page. Also, it is also possible to find URLs that are only discoverable thorough executing a sequence of events. This means that either crawling technique may create new tasks for the other; during URL-based crawling the crawler can discover pages with enabled events and during event-based crawling the crawler can discover new URLs.

In this thesis, we focus on event-based crawling. So, the crawling strategies explained in this thesis should be applied to each distinct URL in an application for a complete coverage. Combining URL-based crawling and event-based crawling is not addressed in this thesis. Simple strategies for such a combination can easily be defined.

Transfer Sequences and Resets

While crawling a RIA, a state usually has to be visited multiple times (at least, once for each enabled event in the state). Unlike traditional applications, the crawler cannot jump to any state in RIAs: to move from the current state to another state, the crawler has to follow a path of already explored events that is known to lead to the desired state. We refer to such a sequence as a *transfer sequence*[1].

> *Transfer Sequence*: A transfer sequence is a sequence of already explored events executed by the crawler to move from one known state to another.

In some cases, the crawler executes a transfer sequence after a *reset*.

> *Reset*: A reset is the action of going back to the initial state by loading the URL.

Sometimes, resetting may be the only option to continue crawling of a URL: a state where there is no enabled event can be reached, or the crawler needs to transfer to a state which is only reachable through the initial state of the URL and the crawler has not yet discovered a transfer sequence to the initial state from the current state.

1.3.6 Crawling Strategy

A crawling strategy is an algorithm that decides how crawling proceeds. That is, for URL-based crawling, the crawling strategy basically decides the exploration order of the discovered URLs. In the case of event-based crawling, the crawling strategy decides the exploration order of the (state, event) pairs (i.e., from which state which event should be explored next).

Effect of Strategies on Crawling Efficiency

We defined crawling efficiency as the ability to discover as much state as possible as soon as possible. The effect of the crawling strategy on the crawling efficiency is significantly different for traditional applications and RIAs. In traditional applications, the crawling

[1] One may ask why the crawler needs to execute a transfer sequence instead of storing each DOM it discovers and simply re-loading the stored DOM of the desired state. However, this is not feasible: this requires the crawler to allocate a significant amount of storage to store the DOMs, and more importantly, in most RIAs, when a stored DOM is loaded to the memory, the functionality of the application will be broken since the JavaScript and the server-side context of the application will not be correct.

strategy does not affect the crawling efficiency much. For a given traditional application, the number of distinct URLs that needs be downloaded is fixed and the order in which these URLs are downloaded does not make much difference since it is usually true that each new URL leads to a new state. What the crawling strategy mostly affects in URL-based crawling is the "effectiveness" of crawling which is the ability to discover the "important" pages quickly based on a given importance metric [59, 25] (we discuss the effectiveness in more details in Section 3.2.1).

For RIAs, on the other hand, crawling strategy is an important factor for the crawling efficiency. This is mainly for two reasons:

- Although the number of events to explore in an application is fixed, it is not true that every event exploration discovers a state. In fact, only a minority of the event explorations lead to new states during the crawl, considering there are often significantly more transitions than states. A crawling strategy for RIAs should be able to predict which (state, event) pairs are more likely to discover a state and give them priority.

- The time spent on executing transfer sequences during the crawl also depends on the decisions of the crawling strategy. The state where the next event will be explored is chosen by the strategy. Also, the choices made by the strategy affects how soon the shortest transfer sequence between two states is discovered. The time spent executing the transfer sequences increases the time required to crawl an application. So, crawling strategies should try to minimize this time for efficiency.

Shortcomings of the Standard Crawling Strategies for RIAs

Two existing and widely-used crawling strategies are the *Breadth-First* and the *Depth-First*. Although these strategies work well with traditional applications, they are not efficient for crawling RIAs since they lack the mentioned characteristics of an efficient strategy: Neither strategy has a mechanism to predict which event is more likely to discover a new state. In addition, both strategies explore the states in a strict order which increases the number and the length of transfer sequences used by these strategies. That is, the Breadth-First strategy explores the least recently discovered state first, whereas the Depth-First crawling strategy explores the most recently discovered state first. Note that, exploring a state means to explore every enabled event of the state. That implies, for example, when these strategies explore an event from a state s, and if as a result, another state s' is reached, the crawler needs to transfer from s' to s in order to finish

remaining unexplored events in s (in the case of Depth-First, assume s' is a known state). A more efficient strategy would try to find an event to explore from the current state or from a state that is closer to the current state, rather than going back to the previous state after each event exploration.

1.4 Motivation and Research Question

The lack of adequate techniques and algorithms required for crawling RIAs has important consequences. The inability to crawl RIAs means that the content in these applications are not indexed by search engines, hence cannot be searched by Web users. To our knowledge, none of the existing search engines have the crawling capability for RIAs[2]. Also, without building a model of the application, RIAs cannot be tested automatically. It is also not possible to automatically scan RIAs for security vulnerabilities or usability issues. Today, none of the existing web application scanners has enough crawling capabilities to handle RIAs [15, 32]. This means that the security vulnerabilities and the other issues targeted by these scanners remain undetected when RIA technologies are used in web applications.

As more and more web applications adopt RIA technologies, the need for addressing the problems related to RIA crawling escalate. Also, web authoring tools make it easy to automatically add RIA technologies to websites. As a result, even the simplest applications built without any programming ability by some content editor easily become non-searchable.

In this thesis, we aim to advance the current limited research on crawling RIAs by exploring the possibilities of designing crawling strategies which are more efficient than the Breadth-First and the Depth-First strategies. This is an important research question since the existing research on RIA crawling still relies on these standard strategies which do not have much potential to be efficient.

[2]For example, Google [41] acknowledges its inability to crawl AJAX applications and suggests a method where the web developer has to present the static HTML snapshot of each state reachable by AJAX, when asked by Google. The web developer also has to produce the URLs for the AJAX states by appending the hashbang sign (#!) followed by a unique name for the state to the application's URL and put these URLs somewhere visible to the crawler, such as Sitemap. When the crawler sees such a URL, it understands that this is an AJAX state and asks the server for the static HTML snapshot corresponding to that state. Obviously, this method just makes crawling the responsibility of the web developer and is not a real solution to RIA crawling.

1.5 Overview and Organization of the Thesis

RIAs break away from the concepts on which traditional web applications are based. As a result, traditional crawling algorithms are not sufficient for crawling RIAs. New crawling algorithms and techniques are needed to be able to search, automatically test or analyze RIAs. One important problem that needs to be addressed is designing efficient crawling strategies for RIAs. It seems possible to design crawling strategies more efficient than the standard crawling strategies, the Breadth-First and the Depth-First since these are general algorithms that do not take into account the features of RIAs. Exploring the possibilities of more efficient crawling strategies is the main goal of our research.

In this research, we follow a general approach called *Model-based crawling* to design efficient crawling strategies. Model-based crawling provides some guidelines to design new crawling strategies for RIAs. Using this approach, several new crawling strategies have been introduced. The first model-based crawling strategy is the *Hypercube strategy*. An initial version of the Hypercube strategy is explained in [16, 18] (in Kamara Benjamin's master thesis); however, this initial version has severe limitations that make it impracticable. For this reason, in this thesis we introduce a new version of the Hypercube strategy that removes these limitations. The second model-based crawling strategy is the *Menu strategy* that is introduced in [26] (in Suryakant Choudhary's master thesis). For the Menu strategy, we provide a short overview. The most recent model-based crawling strategy is the *Probability strategy* which is introduced in this thesis.

In this research, we are collaborating with IBM®. We have implemented our model-based crawling strategies and the other known strategies on a prototype of IBM® Security AppScan® [44], a web application scanner. We show by experimental results that the model-based crawling approach results in more efficient crawling strategies.

1.5.1 Contributions

The main contributions of this thesis can be listed as

- introduction of the concepts of model-based crawling in a formal way,
- a significantly improved version of the Hypercube strategy whose initial version was introduced in [16],
- introduction of the Probability strategy as a new crawling strategy for RIAs,
- implementation of a prototype crawler for RIAs,

- an experimental study where the performances of the existing and the model-based crawling strategies are evaluated using five real AJAX-based RIAs and three test applications,
- an experimental study where several alternative versions of the Probability strategy are evaluated.

1.5.2 Organization

This thesis is organized as follows.

In Chapter 2, we explain our working assumptions and the challenges related to RIA crawling.

In Chapter 3, we present an overview of the work related to traditional crawling and a survey of the existing work related to RIA crawling.

In Chapter 4, we explain the model-based crawling approach and the first model-based crawling strategy, the Hypercube strategy.

In Chapter 5, we first give a short overview of the Menu strategy [26] and then explain the Probability strategy in details.

In Chapter 6, we present the details of our crawler implementation.

In Chapter 7, we present an experimental study that compares the performances of the model-based crawling strategies and the existing strategies on several real and test applications.

In Chapter 8, we conclude the thesis and provide some future directions to expand the research on RIA crawling.

In Appendix A, we present some further experimental results with the alternative versions of the Probability strategy.

Chapter 2

Working Assumptions and Challenges

2.1 Introduction

Although designing efficient crawling strategies is an important problem, it is not the only challenge for crawling RIAs. In this chapter, we explain other challenges related to crawling RIAs.

In Section 2.2, we explain our working assumptions. The challenge to determine if a DOM instance is equivalent to another DOM instance that was seen before is explained in Section 2.3 . The challenge of identifying events is explained in Section 2.4. (The algorithms we have used in our implementation to address these two challenges are explained in Chapter 6.) In Section 2.5, we explain the notion of "intermediate states" caused by AJAX requests. Finally, in Section 2.6 we conclude the chapter.

2.2 Working Assumptions

When building a model, we make some limiting assumptions regarding the behavior of the application being crawled. The assumptions we make are in line with the ones made in the related works. These assumptions are related to determinism and user inputs.

Determinism

We assume that the behavior of the application is deterministic from the point of view of the crawler: from the same state, executing the same event leads to the same state[1]. Formally, the following is satisfied

$$\forall s_x, s_y \in S \, \forall e \in \Sigma. \; s_x = s_y \wedge \delta(s_x, e) = s_k \wedge \delta(s_y, e) = s_l \Rightarrow s_k = s_l \tag{2.1}$$

Similarly, a reset is assumed to always lead to the same initial state. With this assumption, we can use partially extracted model to generate transfer sequences that can be reliably used to move from one known state to another known state.

A dependence of the application that is not directly observable on the client-side can potentially violate this assumption. For example, if there is a change on the server-side state, the result of executing an event from a given state could be different at different times. The state change on the server-side could be a result of some action we do during crawling, such as executing an event that modifies the application database, or it can even be an external factor not related to crawling, such as an application that behaves differently according to time of the day. Since the crawler only observes the changes on the client-side, a possible change on the server-side may not be detected.

How to cope with the cases when the determinism assumption is violated is not addressed in this thesis.

User Inputs

The second working assumption is about user inputs. User inputs are considered an event during crawling. However, there is a very large number of possible values that can be entered by a user (for example, consider the text that can be entered in a text field of a form), so it is not usually feasible to try all of them during the crawl. Instead, we assume that the crawler is provided with a set of user inputs to be used. We are not making any assumptions regarding the coverage of the provided set; we just guarantee that the model that is being built is complete for the values provided.

How to a choose a subset of the possible user inputs that will provide a good coverage is not addressed in this thesis. There has been some research addressing this problem [58, 67, 48]. In the ideal case, the subset provided to the crawler must be enough to discover all the states of the application.

[1] Because of this assumption, it is possible to represent the model we are building as a deterministic Finite State Machine: δ is a function.

2.3 DOM Equivalence

To be able to build a model, the crawler must decide after each event exploration whether the DOM it has reached corresponds to a new state or not. This is needed to avoid exploring the same states over and over again. Moreover, if the current DOM is not a new state, the crawler must know which of the known states it corresponds to.

A simple mechanism is equality where two DOMs correspond to the same state if and only if they are identical. But, equality is a very strict relation and not very useful for most applications. Web pages often contain parts that change when the page is visited at different times or that do not contain any useful information (for the purpose of crawling). For example, if the page contains timestamps, counters, or changing advertisements, using equality will fail to recognize a page when the page is visited at a later time, simply because these "unimportant" parts have changed (see [27] for a technique that aims at identifying the non-relevant parts in a web page automatically).

More generally, the crawler could use a DOM Equivalence Relation[2]. A DOM equivalence relation partitions the DOMs into equivalence classes such that each equivalence class represents a state in the model. Using the DOM equivalence relation, the crawler decides if the current DOM maps to an existing state in the model or not.

The choice of a DOM equivalence relation should be considered very carefully since it affects the correctness of the produced model and the efficiency of the crawl. If the equivalence relation is too strict (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 states that are merged while, in reality, they should be considered different, leading to an incomplete, simplified model.

Unfortunately, it is hard to propose a single DOM equivalence relation that can be useful in all situations. The choice of the DOM equivalence depends on the purpose of the crawl, as well as the application being crawled. For instance, if the purpose of the crawl is content indexing, then the text content of pages should be taken into account. But, in the case of security analysis, the text content usually has no significance for deciding the equivalence of DOMs.

[2]Mathematically, a binary relation, $\sim$, on a set, A, is an equivalence relation iff it has the following three properties: 1. reflexivity ($\forall a \in A.\, a \sim a$), 2. symmetry ($\forall a, b \in A.\, a \sim b \Rightarrow b \sim a$), 3. transitivity ($\forall a, b, c \in A.\, a \sim b \wedge b \sim c \Rightarrow a \sim c$). An equivalence relation partitions the underlying set, i.e., divides the set into non-empty, disjoint subsets whose union cover the entire set. Each subset in the partition is called an equivalence class.

For the correctness of the model produced, it is important to have a DOM equivalence relation that is an equivalence relation in mathematical sense (i.e., the relation must be reflexive, symmetric and transitive). In addition, it is reasonable to constrain the equivalence relation such that the DOMs in the same equivalence class have the same set of enabled events. Otherwise, two equivalent states would have different ways to leave them. This will result in a model that cannot be used reliably to move from one state to the other. When two DOMs with different set of events are mapped to the same state, we can never be sure which set of events we are going to find in that state when we visit it again.

When implementing a DOM equivalence relation, it is important to use an efficient mechanism to decide the equivalence class of a given DOM. It is usually not feasible to store discovered DOMs and compare a given DOM against all. For this reason, fingerprinting techniques are usually used to determine the equivalence class of a DOM. That is, when a DOM is reached, it is first transformed into a normalized form (for example, by removing unimportant components of the DOM) and the hash of this normalized DOM is produced. This hash value is stored and used to identify equivalent DOMs efficiently: if two DOMs have the same hash values then they are considered equivalents.

We note that DOM equivalence is a concept independent of the crawling strategy; a crawling strategy can work with any appropriate DOM equivalence relation.

2.4 Event Identification

Another challenge in crawling RIAs is identification of events. The crawler should be able to detect the enabled events at a state and produce identifiers to differentiate between these events. The event identification mechanism must be deterministic. That is, for an event at a state, the same event identifier must be produced every time the state is visited. This is required since the crawler must know whether an event has already been explored from the state or not. Also, to be able to trigger a known transition at a later time, the crawler needs to recognize the event that corresponds to the transition among the events enabled at the source state. The event identification is also important for DOM equivalence since we require that two equivalent DOMs need to have the same set of enabled events.

In addition, an event identification mechanism should allow the crawler to recognize the instances of the same event at different states. Although it is still be possible to crawl an application without this capability, this is important for designing efficient crawling

strategies. Recognizing instances of the same event at different states allows crawling strategies to make predictions about the event's behavior.

Since events are associated with DOM elements, the problem of event identification is related to unique identification of DOM elements. This is challenging since it is difficult to identify a single solution that would work for all applications. One may suggest using the path of an element from the root node in the DOM tree as an identifier, but this path changes if the place of an element changes in the tree. Similarly, one may be tempted to use the *id* attributes of the DOM elements, but this is not a complete solution on its own. This is because there can be elements with no id assigned. Moreover, although the ids of the elements in a DOM are supposed to be unique, there is no mechanism to force this. It is still possible to assign the same id to multiple elements in the same DOM. Also, there is no requirement for ids to be consistent across different DOMs. Generating the event identifier based on a combination of information about an element such as the values of some selected attributes, the number of attributes and the element type can be possible, but in this case the question of which attributes to include/exclude becomes important.

Like DOM equivalence, event identification should be considered independent of the crawling strategy since a strategy works with any appropriate event identification mechanism.

2.5 Intermediate States

In [16], Benjamin points to the notion of intermediates states caused by the event executions that make AJAX requests. When an event is executed, the crawler usually considers only the state before the event execution and the state reached after the event execution. However, when an event makes an AJAX call, the application actually may go through at least three states: the state before the event execution, the state after the AJAX request is made but before the response is processed completely, and the state after the AJAX response is processed. The number of intermediate states may increase when multiple AJAX requests interleave. Although we do not capture intermediate states in our crawler, it can be useful to consider such states for security testing.

2.6 Conclusion

In this chapter, we explained our working assumptions for building models of RIAs. These are the assumptions that the application is deterministic from the point of view of the crawler and that the crawler is provided with a set of user inputs to be used during the crawl. Although a limiting one, the determinism assumption is a common assumption made in all related works, which are surveyed in the next chapter. The selection of user inputs to be used during the crawl is not addressed in this thesis.

In addition, we have explained the challenges of determining equivalent DOMs, identifying events and the intermediate states caused by AJAX requests. The algorithms used in our crawler to address the first two of these challenges are explained in Chapter 6. Currently, the intermediate states are not captured by our crawler.

Chapter 3

Literature Review

3.1 Introduction

Crawling traditional applications is a well-studied problem with many proposed solutions. The research on traditional crawling goes beyond the basic task of discovering pages; many different research areas are explored, such as defining page importance metrics, maintaining content freshness, politeness, distributed crawling and so on. However, the research on crawling RIAs is more recent and still tries to address the fundamental problem of discovering pages. The majority of the published works on crawling RIAs use one of the standard strategies (Breadth-First and Depth-First); however, these are not efficient strategies for crawling RIAs for the reasons explained previously (in Section 1.3.6).

In Section 3.2, we present an overview of the concepts introduced by the research on crawling traditional web applications . In Section 3.3, we survey the existing research on crawling RIAs and related problems.

3.2 Traditional Crawling

For traditional crawling, Olston and Najork provide a survey [59]. In this section, we briefly present some of the important research areas related to traditional crawling.

3.2.1 Crawling Strategies (URL Ordering)

Crawling strategies for traditional crawlers determine an effective crawl ordering for the discovered URLs. That means "important" pages should be downloaded first. Of course, the importance of a page may depend on the purpose of crawling. For example, in the case of "scoped crawling" [59], the aim is to find the pages that are related to a particular category such as a topic (pages about gardening), a language (pages that are in a given language), a geographical location and so on. In these cases, the importance metric for the crawl also depends on the scope.

For general purpose crawling, a commonly used importance metric is PageRank [61]. PageRank is a connectivity-based metric that defines the importance of a page recursively. The PageRank of a page P is computed based on the PageRank values of the pages that contain links to P. P's importance is higher if many important pages contain links to P. PageRank is also used to measure the effectiveness of crawling strategies. A strategy is said to be more effective if it downloads the pages that have high PageRank values earlier in the crawl.

Some of the general purpose crawling strategies for traditional crawling are the following:

- Breadth-First Strategy: URLs are explored in a breadth-first manner: URLs are visited in the order they are discovered. The biggest advantage of the Breadth-First strategy is its simplicity.

- Back-Link Count: In this strategy, the next URL to explore is the one that has the most incoming links among the pages that are already downloaded [25]. This strategy uses the number of incoming links of a page as a simple estimation of its PageRank.

- Partial PageRank: This strategy is based on the estimation of the PageRank values of unexplored URLs based on the discovered pages so far. The next URL to explore is the one with highest PageRank in the partial model.

Authors of [25] presented experiments showing that the Partial PageRank strategy is more effective than the other two. In [57], another study is presented using a larger dataset to see the effectiveness of Breadth-First strategy (but without comparing with the other strategies). They claimed that Breadth-First is able to find the important pages first with feasible computational cost since continuously calculating Partial PageRank

during the crawl could be infeasible for large datasets. To address this issue an online approximation algorithm is proposed in [1].

We would like to state that, except for the Breadth-First, the mentioned strategies do not make much sense when they are applied directly to RIAs. This is because, these strategies are based on the assumption that any page can be referenced from any other page on the Web. In RIAs, except for the initial state, the states that are reached only by event executions are not referenced from other pages on the Web.

3.2.2 Page Freshness

Another consideration in crawling is maintaining the freshness of the discovered pages. That is, once pages are discovered; revisiting the discovered pages periodically helps keeping the content up-to-date. Page revisits are also useful to detect removed or added links; this reveals the removed pages and leads to discovery of new pages that may remain unknown otherwise. In addition to the crawling strategy, traditional crawlers often have a page revisit strategy that decides how often a page should be revisited.

There are studies that aims at maximizing the average freshness of a collection of pages [28, 24]. Solving this problem requires to solve 3 sub-problems [59]; 1) Deciding on a statistical model that estimates the change frequency of each page, 2) Deciding on target re-visitation frequencies for each page, given the download rate of the crawler (how many pages crawler can download in a second), 3) Deciding on the re-visitation schedule for the pages such that the target re-visitation frequencies can be realized as much as possible.

Of course, a freshness metric is also required. For example, [28, 24] studied the problem under a binary freshness metric where a page is either fresh or not. Authors in [24] also introduced a temporal (continuous) freshness metric where the freshness of a page is evaluated based on the amount of time passed since the first change that has caused the cached copy to become out-dated. In this latter model, the more the cached page remains different from the live page, the less fresh it becomes.

It should be noted that page freshness is not exactly related to obtaining a model of an application. Page freshness is mostly important for search engines to increase the quality of search results by indexing the latest content.

3.2.3 Politeness

Since crawlers are automated tools, they are capable of generating large number of requests in a short amount of time. If care is not taken, the high rate of requests sent to a web server degrades the performance of the server for the regular users. These may even be considered a *denial-of-service attack*[1] by the server and the crawler's access can be blocked as a result.

Overloading a server with high rate of requests is regarded as being "impolite" and crawlers often apply a politeness policy to avoid this. A common approach is to put a delay between successive requests to the same server [59]. Some of the works use a fixed delay such as 10 seconds in [24] and 30 seconds in [13], whereas the authors of [57] use an adaptive approach where the delay is chosen to be proportional to the time it took to download the last page from the server. With the adaptive delay approach, the servers with poor performance are given longer delays.

An additional measure to increase politeness is the robot exclusion method [45]. In this method, a publicly accessible file is used by the websites to specify what pages crawlers are allowed to download. Thus, limiting the load created by the crawlers. Also, in the case of page revisits for freshness, the "if-modified-since" HTTP header is often used to generate conditional requests such that the server sends the requested page only if the page has changed since the time provided in the header.

Politeness is an important issue when crawling applications on public domains, and this is usually done by search engines to index content. However, politeness is usually not a big concern when building a model of the application for testing since it is often possible to crawl an offline instance of the application that is dedicated to testing.

3.2.4 Distributed Crawling

For applications that have a large number of pages (or for Web crawling), using a single process for crawling does not provide enough download rate. The research on distributed crawling of traditional applications aims at increasing the download rate by using multiple crawlers in parallel, while trying to minimize the coordination overhead between the crawlers. The coordination of the crawlers is needed to prevent downloading the same page by multiple crawlers.

Distribution of the work among multiple crawlers is usually done by partitioning the URL space and assigning each crawler a different subset of URLs. In [23], two approaches

[1] A denial-of-service attack is defined as an attempt to make a resource unavailable to its users.

for the assignment of the URLs to the crawlers are explained. The first approach is to use dynamic assignment where a centralized coordinator decides during the crawl which crawler is responsible for a URL. The second and more common approach is to use static assignment where a hash function is used to map each URL to a crawler. This partitioning could be done on different ways such as based on the host names or based on the complete URLs [47]. The geographical locations of the crawlers can also be taken into account for partitioning in order to assign a host to a crawler that is geographically close to the host [36]. Another consideration for partitioning is to minimize the work that needs to be redone if one of the crawlers disappears or a new one joins [20].

Since these partitioning mechanisms rely on URLs, they will not be effective enough for distributing the work when crawling a RIA. It is normal to expect that a RIA has few URLs and under each URL many pages can be reached by executing events. New partitioning algorithms are needed in order to efficiently crawl a RIA in a distributed manner.

3.2.5 Eliminating Redundant and Non-Relevant Content

Sometimes, a page in a web application might be addressed by more than one URL. There are several reasons that may cause this redundancy, such as using URL redirections, adding session identifiers to URLs to keep track of user sessions (URL-rewriting method), having multiple DNS names for the same server, or URL parameters that change how the page looks without changing the content. It is beneficial for a crawler to detect such URLs that lead to identical (or very similar) pages before downloading their content. There has been some research proposing techniques to detect such URLs [14, 29, 2]. These techniques examine the web server logs (or previous crawl logs) and try to learn the rules that can be used to convert each URL to a normalized form. The crawler can then use these rules to detect and eliminate redundant URLs.

A related area is to detect pages that are almost identical among downloaded pages, so called near-duplicate web documents (see [46] for a survey). Near-duplicate web documents are documents that are exactly the same in terms of their main content but differ in small portion of the documents, such as advertisements, timestamps and counters. The main motivations for this research are (a) increasing the quality of content searching and (b) reducing space requirements of search engines. Although it is a valid question to ask if these methods could be used to improve DOM equivalence relations for RIA crawling, the answer is not always positive. Most of these methods work in

batch mode (once documents have already been discovered) and may not be efficient enough for the use during crawling. Also, these methods are based on calculating a similarity measure between two documents; hence, they are not equivalence relations in the mathematical sense.

In [27], Choudhary et al. present a technique to detect non-relevant content, such as advertisements and timestamps, by loading the page twice and comparing the two documents. The parts that have changed between the two loads are likely to be non-relevant. In addition, they propose to use a same technique to detect session identifiers in URLs.

3.3 RIA Crawling

Although limited, there has been some research focusing on crawling of RIAs. Except our research, the main body of work related to crawling RIAs originates from three main sources: a research group in ETH Zurich [33, 51, 38], the research group developing the tool called *Crawljax* [53, 55], and the research group of Amalfitano et al. [4, 6, 5, 7]. The research from ETH focuses on making AJAX applications searchable by indexing their content. The tool Crawljax aims at crawling and taking a static snapshot of each AJAX state for indexing and testing. Amalfitano et al. use execution traces obtained from AJAX applications for automated testing. Below, we give a summary of the techniques used in these works as well as other related work.

3.3.1 Crawling Strategy

To our knowledge, there has not been much attention on the efficiency of crawling strategies. Except for the model-based crawling strategies introduced by our research group and the greedy strategy introduced in [62], the existing approaches use either a Breadth-First or a Depth-First crawling strategy. For this reason, they are limited in terms of strategy efficiency.

In [33, 51, 38], the Breadth-First crawling strategy is used. As an optimization, they propose to reduce the communication costs of the crawler by caching the JavaScript function calls (together with actual parameters) that result in AJAX requests and the response received from the server. If a function call with the same actual parameters is made in the future, the cached response is used, instead of making a new AJAX call.

Crawljax [53, 55] extracts a model of the application using a variation of the Depth-

First strategy. Its default strategy only explores a subset of the events in each state. This strategy explores an event only from the state where the event is first encountered. The event is not explored on the subsequently discovered states. This default strategy may not find all the states, since executing a fixed event from different states may lead to different states. However, Crawljax can also be configured to explore every enabled event at each state; in that case, its strategy becomes the standard Depth-First crawling strategy.

Amalfitano et al. [4, 6, 5] focus on modeling and testing RIAs using execution traces. Their initial work [4] uses execution traces obtained from user-sessions (a manual method). Once the traces are obtained, they are analyzed and an FSM model is formed by grouping together the equivalent user interfaces according to an equivalence relation. In a later paper [6], they introduced a tool, called *CrawlRIA*, which automatically generates execution traces using a Depth-First strategy. That is, starting from the initial state, events are executed in a depth-first manner until a DOM that is equivalent to a previously visited DOM is reached. Then, the sequence of states and events is stored as a trace in a database, and after a reset, the crawl continues from the initial state to record another trace. These automatically generated traces are later used to form an FSM model using the same technique that is used in [4] for user-generated traces.

In [18, 16], Benjamin et al. present the initial version of the first model-based crawling strategy: the Hypercube strategy. This strategy makes predictions by initially assuming the model of the application to be a hypercube structure. This initial version of the strategy has performance drawbacks which prevent it to be applicable even when the number of events in the initial state is as few as 20. Part of this thesis presents a new strategy that uses the same assumptions to make predictions but can be run without any performance issues. In [26], another model-based strategy, called the Menu strategy, is introduced. This strategy is optimized for the case when all instances of the same event lead to the same state (an overview of this strategy can be found in Section 5.2).

In [62], the authors suggested to use a greedy strategy. That is, the strategy is to explore an event from the current state if there is an unexplored event. If the current state has no unexplored event, the crawler transfers to the closest state with an unexplored event. They also suggested two other variations of this strategy. In these variations, instead of the closest state, the most recently discovered state and the state closest to the initial state are chosen when there is not any event to explore from the current state. They experimented with this strategy on simple test applications using different combinations of navigation styles to navigate a sequence of ordered pages. The navigation

styles used are previous and next events, events leading to a few of the preceding and succeeding pages from the current page, as well as the events that lead to the first and last page. They concluded that all three variations of the strategy have similar performance in terms of the total number of event executions to finish the crawl.

3.3.2 DOM Equivalence and Comparison

In [33, 51, 38], the equality is used as the DOM equivalence method. Two states compared based on "the hash value of the full serialized DOM" [38]. They admit that the equality is too strict for DOM equivalence and may lead to too many states being produced.

Crawljax [53] 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 DOM corresponds to a different state than the previous one. If the distance is below some given threshold then the current DOM is considered equivalent to the previous one. Otherwise, the current DOM is hashed and its hash value is compared to the hash values of the already discovered states. Since the notion of distance is not transitive, it is not an equivalence relation in the mathematical sense. For this reason, using a distance in this way has the problem of incorrectly grouping together the states whose distance is actually above the given threshold. But, in a later paper [55], to decide if a new state is reached, the current DOM is compared with all the previously discovered states' DOMs using the mentioned distance heuristic. If the distance of the current DOM from each seen DOM is above the threshold, then the current DOM is considered as a new state. Although this solves the mentioned problem with the previous approach, this method may not be as efficient since it requires to store all the discovered DOMs and compute the distance of the current DOM to each of them.

In [5], Amalfitano et al. proposed DOM equivalence relations based on comparing the set of elements in two DOMs. According to this method, two DOMs are equivalents if both contain the same set of elements. This inclusion is checked based on the indexed paths of the elements, event types and event handlers of the elements. They have also introduced two variations of this relation. In the first variation, only visible elements are considered, and in the other variation, the index requirement for the paths is removed.

3.3.3 Parallel Crawling

To date, we are not aware of any published algorithm for distributed crawling of RIAs. Some authors extended their existing sequential algorithms by running several crawling

instances in parallel. In [51], the authors propose using multiple crawlers on RIAs (or on Web crawling) that use hyperlinks together with events for navigation. The suggested method first applies traditional crawling to find the URLs in the application. After traditional crawling terminates, the set of discovered URLs are partitioned and assigned to event-based crawling processes that run independent of each other using their Breadth-First strategy. Since each URL is crawled independently, there is no communication between the crawlers.

In [55], the authors of Crawljax proposed using multiple threads for speeding up event-based crawling of a single URL application. The crawling process starts with a single thread (that uses a Depth-First strategy). When a thread discovers a state with more than one event, new threads are initiated that will start the exploration from the discovered state and follow one of the unexplored events from there (again using a Depth-First strategy).

3.3.4 Automated Testing

The Crawljax group also published research regarding the testing of AJAX applications: [54] focuses on invariant-based testing, [19] focuses on the security testing of interaction among web widgets, and [64] focuses on the regression testing of AJAX applications.

In [6], Amalfitono et al. compared the effectiveness of different methods to obtain execution traces (user generated, crawler generated and the combination of the two), and the existing test case reduction techniques based on measures such as state coverage, transition coverage and detecting JavaScript faults. In [7], the same authors used invariant-based testing approach to detect faults visible on the user-interface (such as invalid HTML, broken links, and unsatisfied accessibility requirements), in addition to JavaScript faults (crashes) which may not visible on the user-interface but cause faulty behaviour.

In [50], Marchetto et al. use a state-based testing approach based on a FSM model of the application. Their model construction method uses static analysis of the JavaScript code and dynamic analysis of user session traces. They try to reduce the size of the models using abstraction of the DOM states, rather than using DOM states directly in the model, and this may require some manual activity to ensure correctness. Based on this model, they produce test sequences that contain "semantically interacting events". (Two events are semantically interacting if their execution order changes the outcome.) In [49], they proposed search-based test sequence generation using hill-climbing, rather

than exhaustively generating all the sequences up to some maximum length.

3.3.5 Ranking (Importance Metric)

In [38], the authors propose a ranking mechanism for the states in RIAs. The aim is to assign an importance value to states to determine their rank in search results. The proposed mechanism, called *AjaxRank*, is an adaptation of PageRank [61]. Similar to PageRank, AjaxRank is connectivity-based, but instead of hyperlinks, the transitions are considered. AjaxRank gives more importance to the initial state of the URL (since it is the only state reachable from anywhere directly); hence, the states that are closer to the initial state also get higher ranks.

In addition to ranking the results of a search query, the importance metrics for RIAs can be useful for assessing the effectiveness of crawling strategies and help designing more effective crawling strategies for RIAs.

3.3.6 Related Graph Problem

Based on our definition of efficiency, the problem of designing an efficient strategy for RIAs can be considered as a graph exploration problem. That is, the aim is to visit every node at least once in an "unknown" directed graph by minimizing the total sum of the weights of the edges traversed. The offline version of this problem, when the graph is known beforehand, is called the Asymmetric Traveling Salesman Problem (ATSP) which is NP-Hard.

There are some approximation algorithms for different variations of the unknown graph exploration problem [52, 31, 39, 37]. However, we do not know any which suits to our case. Rather than trying to adapt these algorithms to our case, we design our own algorithms for crawling RIAs since there would be no theoretical improvement. There is no general algorithm with a constant competitive ratio[2]. The authors in [52, 31, 37] consider undirected graphs, and in [37], it has been shown that the lower-bound on the competitive ratio is n^2 for undirected graphs where n is the number of nodes in the graph. In [39], the authors consider directed graphs, but they make the following assumption: when the agent (the crawler in our case) visits a node, it also knows the destinations of the outgoing edges of the node. Since, when crawling a RIA, it is not possible to know the destination of an unexplored event beforehand, this assumption is not valid in our

[2]Competitive ratio is the ratio of the cost of the solution produced by an approximation algorithm and the cost of an optimal solution.

case. When this assumption is made, it has been shown that the lower bound on the competitive ratio for a deterministic algorithm is $(n - 1)$ [39].

3.4 Conclusion

The problem of traditional crawling is well-studied and many solutions have been proposed regarding different aspects of crawling. Discovering the pages in a traditional application is not a difficult task since even a simple Breadth-First strategy is efficient enough. For traditional applications, the effectiveness of crawling strategies are studied which means to discover important pages first. In addition, techniques are proposed to address the issues of maintaining page freshness and eliminating duplicate/irrelevant content. The concept of politeness is used to prevent the crawler from overloading a public web server with the large amount of requests generated during the crawl. Distributed crawling algorithms are proposed to reduce the crawling time by using multiple crawlers.

Compared with the research on crawling traditional applications, the research on crawling RIAs is very recent. The problem of building a model for a RIA has still not been addressed completely. Although this problem is not solved efficiently by using a Depth-First or a Breadth-First strategy, these are the most common strategies used so far in the existing research.

Chapter 4

Model-based Crawling

4.1 Introduction

In event-based crawling, the aim is to start from a given URL and extract a model of the states that are reachable from the initial state of the URL through event executions. Our goal is to extract this model "efficiently", that is, to find all the states as quickly as possible, while being guaranteed that every state will eventually be found (under our working assumptions). Without any knowledge of the RIA being crawled, it seems difficult to devise a general efficient strategy. For example, the Breadth-First and the Depth-First strategies are guaranteed to discover a complete model when given enough time, but they are usually not very efficient as we have explained out in Section 1.3.6. One of their drawbacks is the lack of a mechanism to predict which events are more likely to discover a new state.

To be efficient, a crawling strategy could use an anticipation of the behavior of the application being crawled. If we can identify some general patterns that we anticipate will be found in the actual models of the RIAs being crawled, then these patterns can be used to forecast the model of the application. This idea is the basis of *model-based crawling*, which is the methodology we use to design efficient crawling strategies for RIAs.

In this chapter, we explain model-based crawling and present the Hypercube strategy as the first example of model-based crawling. In Section 4.2, we explain the concepts of model-based crawling. In Section 4.3, we present the model for the Hypercube strategy and give an overview of the initial version of the strategy. In Section 4.4, we introduce an improved version of the Hypercube strategy which overcomes the practical shortcomings of the initial strategy. In Section 4.5, we conclude the chapter.

4.2 Model-based Crawling

4.2.1 Meta-Model

We use the term *meta-model* to represent a class of applications that share certain behavioral patterns. A meta-model, in our context, is a model defining the characteristics of a set of RIA models. The RIA models that follow the characteristics of a meta-model are the instances of the meta-model[1]. A meta-model is defined by specifying the general characteristics of its instances. These characteristics usually capture the relations of the events with the states and with the other events. For example, the characteristics may provide an answer to questions such as: "Is executing a particular event going to lead to a new state?", "Is executing a particular event going to lead to a state where there is a different set of events?" or "Which of the known states will be reached when a particular event is explored?" and so on. In model-based crawling, a meta-model is used as a means to anticipate the model of the application that is being crawled.

4.2.2 The Methodology

In model-based crawling, a strategy is designed based on a chosen meta-model. Such a strategy uses the chosen meta-model as a guide for crawling. That is, we initially assume that the application we are crawling is an instance of the chosen meta-model. Thus, the strategy can be very efficient (possibly an optimal one) for crawling applications that are instances of the chosen meta-model. But, this does not mean that applications that are not instances of the chosen meta-model cannot be crawled. In fact, the actual RIA will in practice almost never be a perfect match for the given meta-model. Model-based crawling must account for the discrepancy between the anticipated model and the actual model, allowing the strategy to adapt to the application being crawled.

To summarize, a strategy is designed in three steps:

1. A meta-model is chosen.

2. A strategy for crawling applications whose models follow the meta-model is designed. Ideally, the strategy must be optimal if the actual model is a perfect match

[1]For example, the meta-model we explain in this chapter is called Hypercube meta-model. There can be different instances of the Hypercube meta-model. The difference would be in the dimensions of the hypercube and in the set of events. The Hypercube meta-model represents the general concept that all the instances of the Hypercube meta-model share: being a hypercube.

for the meta-model.

3. Steps to take are specified in case the application that is crawled deviates from the meta-model.

Actual Model vs. Anticipated Model

In the context of model-based crawling, we often talk about two models: the actual model of the application and the model we are anticipating to find according to the chosen meta-model.

> *Actual Model*: The *actual model* of a given application is the model discovered during the crawl. As defined in Section 1.3.5, we represent the actual model as a graph $G = (V, E)$.
>
> *Anticipated Model*: The *anticipated model* of the application is the model we are anticipating to discover based on the meta-model characteristics. We represent the anticipated model also as the graph, written $G' = (V', E')$.

In model-based crawling, the mechanism for handling the violations of the meta-model always makes sure that the anticipated model conforms to the actual model discovered so far. Hence, the actual model is a sub-graph of the anticipated model. The difference between the two are the anticipated states yet to be discovered V^A and unexplored anticipated transitions E^A. (i.e., $V' = V \cup V^A$ and $E' = E \cup E^A$)

Choosing a Meta-Model

A crucial step in model-based crawling is to find a meta-model that will allow us to anticipate the behavior of the application as accurately as possible. It is a challenge to find a good meta-model that is generic enough to cover most RIAs, but at the same time, specific enough to allow making some valid anticipations. To find a good meta-model, common behaviors that apply to majority of RIAs can be observed and experiments with different meta-models can be done. We discuss a possible solution to the challenge of finding good meta-models as part of the future works in Chapter 8.

Designing an Optimized Strategy

When designing a strategy for a given meta-model, we often use a two-phase approach:

- *The State Exploration Phase* is the first phase that aims at discovering all the states that are anticipated by the meta-model as efficiently as possible. Given the extracted model so far and assuming that the unexplored parts of the application will follow the meta-model anticipations, it is possible to know whether there are any more states to be discovered. Once, based on these anticipations, it is decided that there is not any new state to discover, the strategy moves on to the second phase.

- *The Transition Exploration Phase* is the second phase that explores the events that has not been explored yet. In the state exploration phase, the crawling strategy does not necessarily explore every event; in this first phase, the strategy only explores the events which it anticipates will help discovering new states. However, we cannot be sure that we have discovered all the states unless each event is explored. If a new state is discovered in the transition exploration phase or the strategy anticipates that more states can be discovered, then the strategy switches back to the state exploration phase.

Handling Violations

During the crawl, whenever a discrepancy between the actual model and the anticipated model is detected, the anticipated model and the strategy are revised according to the actual model uncovered so far. There may be different ways to achieve this, but one simple and consistent mechanism is to assume that the characteristics of the meta-model will still be valid for the unexplored parts of the application. So, using the same characteristics, a strategy can be obtained for the unexplored parts as it was done initially.

4.3 Hypercube Meta-Model and the Initial Strategy

In the following, we present the Hypercube meta-model and an optimal strategy to crawl the instances of this meta-model. The idea of using hypercube as a meta-model for crawling RIAs was introduced in [17]. In [16, 18], a complete crawling strategy based on the hypercube meta-model was given. However, in this initial version of the strategy, the actions that should be taken has to be precomputed ahead of time. The pre-computation overhead prevented this initial algorithm to be applicable even for very small applications. In this thesis, we present a new algorithm that addresses this issue completely.

In this section, we first give an outline of the Hypercube meta-model and the initial version of the Hypercube strategy. In the next section, we explain the new algorithm and prove that this strategy is an optimal crawling strategy for the Hypercube meta-model.

4.3.1 Hypercube Meta-Model

As its name suggests, the Hypercube meta-model is the class of models that have a hypercube structure. The Hypercube meta-model is formed based on two assumptions:

- *A1:* The events that are enabled in a state are pair-wise independent. That is, in a state with a set of enabled events, executing a given subset of these events leads to the same destination state regardless of the order of their execution.
- *A2:* When an event e is executed in a state s, the set of events that are enabled in the reached state is the same as the events enabled in s minus e.

These assumptions reflect the anticipation that executing an event does not affect (enable or disable) other events and executing a set of events from the same state in different orders is likely to lead to the same state. Based on these assumptions, the initial anticipated model for an application whose initial state has n events is a hypercube of dimension n. Figure 4.1 shows a hypercube of dimension 4. The vertex at the bottom of the hypercube represents the initial state with four events $\{e1, e2, e3, e4\}$ enabled. Initially, the remaining vertices represent the anticipated states. Each vertex is labeled by the events enabled in the state (for readability not all the labels are shown). Each edge is directed from the lower incident vertex to the upper incident vertex and represents an anticipated transition of the application initially.

In a hypercube of dimension n, there are 2^n states and $n \times 2^{n-1}$ transitions. The height of a state in the hypercube is the number of transitions that must be traversed to reach it from the initial state. The set of states in a hypercube of dimension n can be partitioned as $\{L_0, L_1, L_2, \ldots, L_n\}$ where L_i is the set of states of height i. We call L_i the "level i" of the hypercube. L_0, L_n and $L_{\lfloor n/2 \rfloor}$ are called the bottom, the top and the middle of the hypercube, respectively. We refer to all levels higher than the middle as the "upper half" and the levels lower than the middle as the "lower half" of the hypercube.

4.3.2 Violations of the Hypercube Assumptions

When the RIA does not fully follow the hypercube meta-model, we have "violations" of the hypercube assumptions. With this meta-model, there are four possible violations

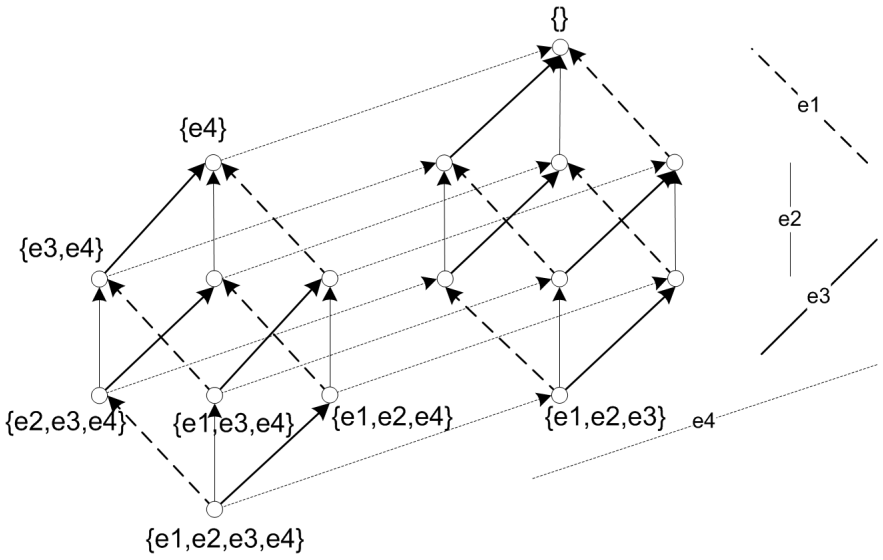

Figure 4.1: A Hypercube of Dimension 4

which are not mutually exclusive. The first meta-model assumption *A1* can be violated in 2 ways:

- *Unexpected Split:* In this case, after executing an event we expect to reach a state that has already been visited, but we actually reach a new state.
- *Unexpected Merge:* In this case, after executing an event we unexpectedly reach a known state (i.e., not the expected known state).

A2 can also be violated in 2 ways:

- *Appearing Events:* There are some enabled events that were not expected to be enabled in the reached state.
- *Disappearing Events:* Some events that were expected to be enabled in the reached state are not enabled.

As explained before, we need to have an efficient strategy that handles all four violations.

4.3.3 The Initial Strategy

For the hypercube meta-model, Benjamin et al. presented an initial strategy [16, 18]. We summarize briefly this initial algorithm and explain its shortcomings addressed in

Section 4.4. These algorithms use the characterization of the hypercube as a partially ordered set to produce an efficient strategy.

A hypercube is the partially ordered set of all subsets of n elements ordered by inclusion. In our case, each subset of the n events found in the initial state represents a state in the hypercube. This characterization is useful for generating an optimal state exploration strategy for a hypercube model. In a partially ordered set, a set of pairwise comparable elements is called a chain. Thus, each directed path in the hypercube is a chain. A set of chains covering every element of the order is known as a chain decomposition of the order. A *Minimum Chain Decomposition (MCD)* of the order is a chain decomposition of minimum cardinality (see [10] for an overview of concepts). For the hypercube model, a minimum chain decomposition is a set A of paths that contain every state in the hypercube such that A is of minimal cardinality with that property (i.e., following an MCD of an hypercube allows us to visit every state in the hypercube using the minimum number of events and the minimum number of resets). In [30], Dilworth proved that the cardinality of any minimal chain decomposition is equal to the *width* of the order, that is, the cardinality of a largest subset whose elements are pairwise non-comparable. In a hypercube of dimension n, the width is the number of states in the middle level which is $\binom{n}{\lfloor n/2 \rfloor}$. A minimum chain decomposition algorithm that can also be used for hypercubes is given in [21]. Each chain produced by this algorithm is of the form $C =< v_i, v_{i+1}, \ldots, v_{i+k} >$ where v_i is a state at level i. In this decomposition, each chain is a unique path in the hypercube, but it does not necessarily start from the bottom of the hypercube. In a chain C, the state at the lowest level is called the "chain starter state" or the "bottom" of C. The set of MCD chains produced for a hypercube of dimension 4 (shown in Figure 4.1) is the following

1. $< \{e1, e2, e3, e4\}, \{e2, e3, e4\}, \{e3, e4\}, \{e4\}, \{\} >$
2. $< \{e1, e2, e3\}, \{e2, e3\}, \{e3\} >$
3. $< \{e1, e2, e4\}, \{e2, e4\}, \{e2\} >$
4. $< \{e1, e2\} >$
5. $< \{e1, e3, e4\}, \{e1, e4\}, \{e1\} >$
6. $< \{e1, e3\} >$

By definition, an MCD provides a complete coverage of the states of a hypercube in an optimal way, that is, using the minimum number of events and the minimum number of resets. Thus, for the state exploration phase, it is enough to generate an MCD of the hypercube. However, an MCD does not cover all the transitions of the hypercube. In the initial hypercube strategy, we devised another algorithm that generates a larger set of chains, called *Minimum Transition Coverage (MTC)*, to cover all the transitions in a hypercube in an optimal way. Since MCD chains already traverses some of the transitions, the MTC algorithm can be constrained with an already generated MCD so that the set of MTC chains contains every MCD chain. The number of paths in an MTC is $\binom{n}{\lfloor n/2 \rfloor} \times \lceil n/2 \rceil$, which is the number of transitions leaving the middle level.

The combination of MCD and MTC provides an optimal way of crawling a RIA that perfectly follows the Hypercube meta-model. That is, for a hypercube, an MTC uses the minimal number of resets and event executions to traverse each transition at least once. Moreover, among MTC chains, the ones that contain the MCD chains are given exploration priority. Thus, all the states of the hypercube are visited first, using the minimal number of events and resets.

The initial algorithm also provides a revision procedure to update the existing set of chains to handle the violations of the hypercube assumptions. The revision procedure basically replaces the chains that become invalid and adds new chains if necessary.

An Example

To better explain the concepts of the anticipated model and the violations of the hypercube assumptions, we provide a simple example. The example details the partial exploration of an example application whose model is shown in Figure 4.2. The initial state of the application, $s1$, has three enabled events $\{e1, e2, e3\}$ and there are 8 states in total.

Figure 4.3 shows (partially) the steps taken by the Hypercube strategy to crawl the application in Figure 4.2. Each diagram in Figure 4.3 shows the current anticipated model: the solid nodes and edges show the actual discovered model, whereas dashed nodes and edges belong to the anticipated model only.

The first diagram, 4.3.(1), shows the initial situation: the only discovered state is $s1$ and the anticipated model is a hypercube of dimension 3 based on $s1$. The MTC chains that are generated for crawling this hypercube are listed below. The highlighted sequences are the MCD chains.

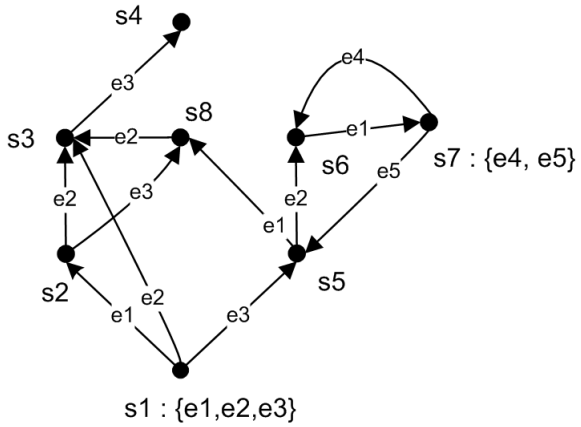

Figure 4.2: Model of the Example Application

1. $<$ **{e1**, **e2**, **e3},{e2**, **e3},{e3},{}** $>$
2. $< \{e1, e2, e3\},$ **{e1**, **e3},{e1}**, $\{\} >$
3. $< \{e1, e2, e3\},$ **{e1**, **e2},{e2}**, $\{\} >$
4. $< \{e1, e2, e3\}, \{e2, e3\}, \{e2\} >$
5. $< \{e1, e2, e3\}, \{e1, e3\}, \{e3\} >$
6. $< \{e1, e2, e3\}, \{e1, e2\}, \{e1\} >$

The Hypercube strategy starts with the execution of the first chain. This is shown in diagrams 4.3.(1)-(4). In this example, the first chain is executed without any violations and the states $s1, s2, s3, s4$ are discovered as anticipated.

4.3.(5) shows the situation after executing the first event ($e2$ from $s1$) of the second chain. In this case, we were anticipating to reach a new state that has enabled events $\{e1, e3\}$, but we have reached an already discovered state ($s3$) that has $e3$ as the only enabled event. This is a violation of both *A1* and *A2* since we have an unexpected merge and a disappearing event. Notice that, after this violation the anticipated model is updated. The state we were expecting to reach is removed since this was the only way to reach it in the model (This also means that the 5th chain is not valid anymore since it was supposed to execute an event from the unreachable anticipated state. So, it has to be removed as well). After this violation, we cannot continue with the current chain; the strategy moves on to the third chain.

4.3.(6) and 4.3.(7) show the execution of the first two events in the third chain. These executions do not cause any violations. However, as shown in 4.3.(8), the last event in the chain causes a violation. We were expecting to reach $s4$ by executing $e1$ from $s6$, but we reached to a new state $s7$ with two enabled events $\{e4, e5\}$. This is a violation of both *A1* and *A2*: it is an unexpected split and there are appearing/disappearing events in the state reached. After this violation, the anticipated model is updated by adding a new hypercube of dimension 2 based on $s7$. That means, new chains have to be computed and added to the set of existing chains, while the current chain becomes invalid.

As the concepts of an anticipated model and the violations are visualized in Figure 4.3, we do not show the further exploration steps that would be performed by the strategy, until all the events are explored.

Shortcomings of the Initial Strategy

Although the initial algorithm is an optimal crawling strategy for applications that follow the Hypercube meta-model, it is not a practical one. The reason is that it requires pre-computing all the chains before the exploration. This is problematic since the anticipated model has a size which is exponential in the number of events enabled in the bottom state. For example, if we attempted to crawl a RIA that has 20 events on its initial page (which is, in fact, a very small number), we could have not even started the exploration since there might not be enough memory to generate the MTC chains (for 20 events, 1,847,560 chains need to be generated). Furthermore, if the application being crawled happens not to fit the hypercube model, then this pre-computation is largely in vain. We would be unable to crawl the RIA simply because we could not generate the strategy. In addition, in case we actually managed to start crawling, maintaining the large number of chains would be very difficult since we have to replace the chains that are not valid anymore.

4.4 The New Hypercube Strategy

In order to execute the Hypercube strategy in an efficient manner, instead of generating the whole strategy before the exploration, the decisions for the next actions can be made on-the-fly, at the time we need them. For state exploration phase, what we need is the ability to identify the successor of a state in its MCD chain. That is, for any state in the hypercube, we need a way to calculate the event to execute to reach the next state

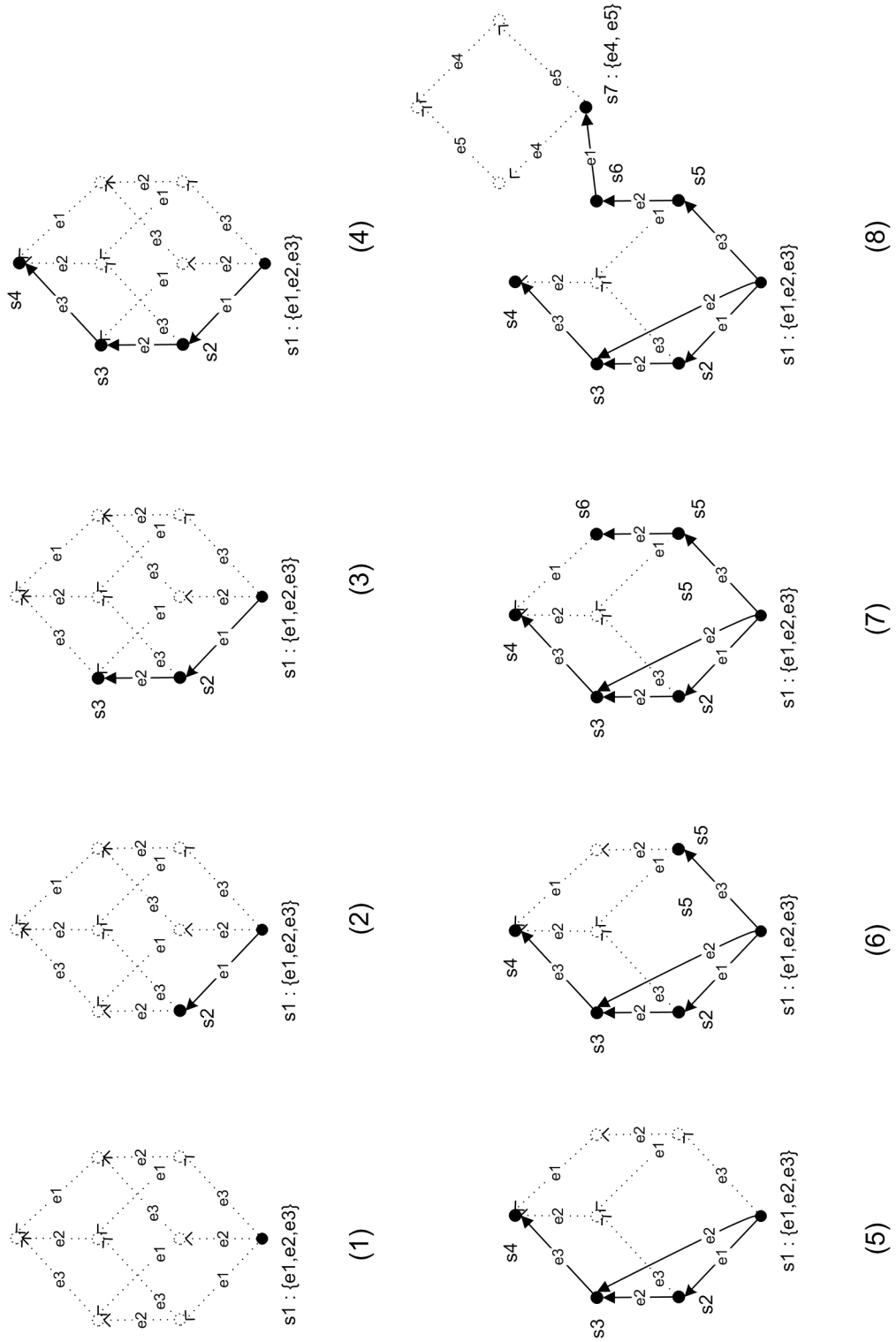

Figure 4.3: (Partial) Crawling of the Example Application

in the MCD chain. Then, we can execute an MCD chain step by step. The optimal transition exploration for hypercube is also possible by a simple greedy approach. Since there are no stored chains to maintain, the revision of the strategy is also easy in case of violations. In the following, we detail this efficient execution of the Hypercube strategy and provide algorithms for this purpose.

Algorithm 1 shows the global variables and the main body of the Hypercube strategy which extracts a model for a given URL. The global variables are listed below.

- v_1 is a vertex representing the initial state. We assume the method `Load` loads the given URL and returns a vertex that represent the initial state.
- $G = (V, E)$ is the extracted model, initially $G = (\{v_1\}, \emptyset)$.
- $G' = (V', E')$ is the anticipated model, initially a hypercube based on v_1. The anticipated model (which can be very large) is not actually constructed in memory. It is merely a hypothetical graph that we use for explanation purposes. The structure of a hypercube allows us to know all the transitions from any state without actually creating the anticipated model.
- $v_{current}$ is a reference to the vertex representing the current state. It is updated after each event execution.
- *phase* shows in which phase the crawling strategy is. It can have one of the three possible values: *stateExploration*, *transitionExploration* or *terminate*. It is initialized to *stateExploration*. Crawling continues until its value becomes *terminate*.

4.4.1 State Exploration Strategy

The optimal state exploration strategy for an hypercube is to follow an MCD of the hypercube. However, we must do it without generating the chains in the MCD ahead of time. The key to achieving this is the ability to determine the state that comes after the current state in an MCD chain, without generating the chain beforehand. For an MCD, we call the state that follows a given state v in the MCD chain as the MCD successor of v and write MCDSuccessor(v). In addition to the successors, we also need to identify from which state an MCD chain starts: the *chain starter*.

Algorithm 1: The Hypercube Strategy

Input: url: the URL of the application to crawl.
Output: $G = (V, E)$: the model of the application
global v_1 := `Load(url)` ;
global $V := \{v_1\}$, $E := \emptyset$;
global $G' := (V', E') :=$ A hypercube graph based on v_1;
global $v_{current} := v_1$;
global *phase* := *stateExploration*;
while *phase* != *terminate* **do**
  if *phase* == *stateExploration* **then**
    `StateExploration`();
  else
    `TransitionExploration`();
  endif
endw

Identifying MCD Successors and Chain Starters

Two different approaches for calculating the MCDSuccessor function are presented in [3], and [42]. (According to [43], both approaches yield the same decomposition that is also produced by the algorithm given in [21]).

The approach explained in [42] is based on parenthesis matching and works as follows: since each state in the hypercube is characterized by a subset of the set of events enabled at the initial state ($\{e_1, e_2, \ldots, e_n\}$), a possible representation of a state v in the hypercube is a n-bits string representation $x_1 x_2 \ldots x_n \in \{0, 1\}^n$ such that the bit x_i is 0 if and only if e_i is enabled at v. To find MCDSuccessor(v), we use this bit string representation of v. We regard each 0 as a left parenthesis and each 1 as a right parenthesis and match the parenthesis in the traditional manner as shown in the Function `MCDSuccessor(`v`)`.

The function keeps track of a set called *IndexesOfUnmatchedZeros*. The set is empty initially and will contain the indexes of the unmatched zeros at the end. The function starts from the leftmost bit x_1 and scans the string such that when a 0 bit is encountered, the index of the 0 bit is added temporarily to the set. When a 1 bit is encountered, it is matched with the rightmost unmatched 0 bit to the 1 bit's left. This is achieved by removing from the set the maximum value. At the end, if the set is empty (i.e., all 0's are matched), then v has no successor. That means, v is the last state in the MCD chain. Otherwise, the minimum index stored in the set is the index of the leftmost unmatched

0 bit. We obtain the bit string of MCDSuccessor(v) by flipping the bit at that position. That means, if i is this minimum index then we have to execute event e_i to reach the MCDSuccessor(v) from v. Using this simple method, an MCD chain can be followed without pre-computation. In addition, the starting states of these MCD chains (i.e., chain starters) are the states whose bit strings do not contain any unmatched 1 bits.

For instance, if the bit string representation of a state v is 1100110001, then we have the following parenthesis representation

```
))(())(((
1100110001
```

where the leftmost unmatched 0 (left-parenthesis) is the seventh bit, so we have to execute the corresponding event (i.e., the seventh event among the events enabled at the bottom of the hypercube) from v to reach the successor of v, MCDSuccessor(v) = 1100111001. In addition, since the bit string of v contains unmatched 1's (the first and the second bits), v is not a chain starter state.

Function `MCDSuccessor(`v`)`

Input: v: a vertex
Output: the MCD successor of v
IndexesOfUnmatchedZeros $= \emptyset$;
Let $x_1x_2 \dots x_n \in \{0,1\}^n$ be the bit string representation of v;
$i := 1$;
while $i \leq n$ **do**
  if $x_i == 0$ **then**
    IndexesOfUnmatchedZeros := *IndexesOfUnmatchedZeros* $\cup\, i$;
  else
    maxIndex := `MAX`(*IndexesOfUnmatchedZeros*);
    IndexesOfUnmatchedZeros := *IndexesOfUnmatchedZeros* $\setminus$ *maxIndex*;
  endif
endw
if *IndexesOfUnmatchedZeros* $== \emptyset$ **then**
  return *nil*;
else
  minIndex := `MIN`(*IndexesOfUnmatchedZeros*);
  bitStringSuccessor := `FlipTheBitAt`($x_1x_2 \dots x_n$, *minIndex*));
  return *the vertex corresponding to bitStringSuccessor*;
endif

Execution of the State Exploration Phase

The Procedure `StateExploration` describes the execution of the state exploration phase. In this phase, we follow the MCD chains one by one using the successor function described above. In order to execute an MCD chain, we first need to find a chain starter state whose MCD chain has not been executed yet. At the very beginning of the crawl, since the initial state is a chain starter state, we can immediately start by executing the corresponding MCD chain. But, if the current state is not a chain starter, we find a path from the current state to the closest chain starter that we have not tried to reach before (notice that, the chain starter state is an anticipated state, so it is possible that the chain starter we are expecting to reach does not exist)[2]. We use the path to attempt to reach to the chain starter. (To execute a path, the function `ExecutePath` is used. This function, which will be given later, executes the events in the path one after the other, if needed updates the actual and anticipated models. If a violation is detected during the execution of the path, `ExecutePath` returns with value false.) If there is no violation, we do reach the chain starter and we start the execution of the corresponding MCD chain. If a violation occurs, we stop the execution of the current chain (which is not valid anymore) and start looking for another chain starter. The state exploration phase finishes when we have tried to execute all MCD chains for the current anticipated model.

Preserving Optimality When executing this strategy, there are some additional steps that we must take in order to preserve the optimality of the Hypercube strategy. First of all, during the state exploration phase, we explore more than the MCD chains. In addition to the MCD chains, whenever possible, we try to traverse unexplored transitions on the path used to reach a chain starter (rather than using already explored transitions). Also, when we come to the end of an MCD chain, we continue exploring transitions, rather than immediately starting the next MCD chain. In other words, the MCD chains are extended towards the bottom and the top using unexplored transitions. Otherwise, these transitions that we have not explored during state exploration while we had the opportunity will cause at least one extra event execution and possibly one extra reset. Moreover, when constructing a path P, we want it to be in the following form: $P = P_P P_S$ where P_P is a (possibly empty) prefix path that consists of already explored transitions

[2]If there are multiple such chain starters, we choose the one whose MCD chain is longer. This is because, a longer MCD chain means more anticipated states to discover. In an n dimensional hypercube, the length of an MCD chain whose chain starter is at level l is given by the formula: $n - 2 \times l$.

Procedure `StateExploration`

while *there is a chain starter that we have not yet attempted to reach* **do**
Let P be a path in G' from $v_{current}$ to the closest such chain starter;
`// try to reach to the chain starter`
if `ExecutePath`$(P) == TRUE$ **then**
`// execute the MCD chain`
$v_{successor}$:= `MCDSuccessor`$(v_{current})$;
while $v_{successor}$! = *nil* **do**
if `ExecutePath`$((v_{current}, v_{successor})) == FALSE$ **then**
break;
else
$v_{successor}$:= `MCDSuccessor`$(v_{successor})$;
endif
endw
`// if the end of chain is reached, extend the chain`
if $v_{successor} ==$ *nil* **then**
while *there is an unexplored transition* $(v_{current}, v'; e)$ **do**
`// explore the event and check for violation`
if `ExecutePath`$((v_{current}, v'; e)) == FALSE$ **then**
break;
endif
endw
endif
endif
endw
phase := *transitionExploration*;

and P_S is a path that contains only previously unexplored transitions. That means, in a path, all the transitions that follow the first unexplored transition should also be unexplored. In particular, the paths that are used to reach a chain starter during the state exploration phase should contain, as much as possible, unexplored transitions and there should not be an already explored transition following an unexplored transition. Based on the same principle, the paths that extend MCD chain toward the top and the paths taken during the transition exploration (the phase explained next) should end when a state without unexplored transitions is reached. In that case, we should go back to the bottom of the hypercube and start exploring a new chain.

4.4.2 Transition Exploration Phase

The Procedure `TransitionExploration` describes the execution of the transition exploration phase. In this phase, we use a simple greedy strategy to explore the remaining unexplored events. The strategy is to always explore an unexplored transition that is closest to the current state. That is, we search in the actual model the shortest transfer path from the current state to a state which has an unexplored event. We reach to that state using the path, execute the unexplored event, and check if the state that is reached violates hypercube assumptions. Any violation is handled as we explain next.

Procedure `TransitionExploration`

while *phase* == *transitionExploration* **do**
  if *there is an unexplored transition* **then**
    Let $(v, v'; e) \in E'$ be the closest unexplored transition to $v_{current}$;
    Let P be a path constructed by appending $(v, v'; e)$ to a shortest path from $v_{current}$ to v;
    `ExecutePath`(P);
  else
    phase := *terminate*;
  endif
endw

The crawl terminates when all the enabled events at each discovered state are explored.

4.4.3 Executing Events, Updating the Models and Handling Violations

Function `ExecutePath(`P`)`

Input: P: a path to traverse

Output: FALSE if any hypercube assumption is violated, otherwise TRUE

if P *requires reset* **then**
  $v_{current}$:= `Load(url);`
endif
foreach *transition* $(v, v'; e) \in P$ *from the first to the last* **do**
  $v_{current}$:= `Execute(`e`);`
  if $(v, v'; e) \in E^A$ **then** `// is this an event exploration?`
    if `Update((`$v, v'; e$`))` $==$ *FALSE* **then**
      return *FALSE*;
    endif
  endif
endfch
return *TRUE*;

To execute transitions, we call the function `ExecutePath`. Given a path, the function executes the sequence of events on the path, possibly after a reset. We assume that a method called `Execute` executes the given event from the current state and returns a vertex representing the state reached. That is, if the event execution leads to a known state, `Execute` returns the vertex of the state. Otherwise, `Execute` creates and returns a new vertex for the newly discovered state (we use the method `Execute` as a notational convenience combining event execution and the DOM equivalence relation).

The path provided to `ExecutePath` may contain both anticipated transitions (i.e., not yet explored) and already explored transitions. The return value of `ExecutePath` shows whether a violation of the hypercube assumptions is detected during the execution of the path. If a violation is detected, the function returns immediately with value false.

After each explored transition, we must update the actual model, check for the violations of the hypercube assumptions, and if needed, update the anticipated model. The function `Update` describes the mechanism to update the models and to handle the violations. The returned value of the function is false when a violation is detected.

The function `Update` is given the transition that has just been explored. The function adds this transition to the actual model. If a new state is reached, it also adds a new state

to the actual model. Then, it checks for a violation. This is checked by the expression $v_{current} != v'$. A violation is detected if the inequality holds. Here, we are checking the inequality of two vertices: one representing the state reached, $v_{current}$ (an actual state), and the other, v', representing the state that we were anticipating to reach. The latter can be representing either an anticipated state or an actual state. The semantics of the comparison is different in these cases. If v' represents an actual state (i.e., we have just explored a transition that was anticipated to connect two known states), then we just check if the vertices represent different states to detect a violation. Otherwise, if v' represents an anticipated state, then the inequality is satisfied only if the reached state is not new or the enabled events on the new state do not satisfy the hypercube assumption *A2*.

When there is a violation, we update the anticipated model, still assuming that the hypercube assumptions (*A1* and *A2*) remain valid for the unexplored parts of the application. For this reason, if we unexpectedly reach a new state, we add a new (anticipated) hypercube to the anticipated model. Notice that, in such a case the phase is reset to *stateExploration*. In case of any violation, we have to remove the anticipated transition and any unreachable anticipated states from the anticipated model (again, since the anticipated model is a hypothetical graph, we do not actually remove anything in the real implementation of the strategy.).

4.4.4 Complexity Analysis

The worst-case time complexity of the Hypercube strategy can be analyzed in terms of the size of the actual model of the application, $|E|$, and the maximum number of enabled events at a state, n (i.e., n is the maximum outdegree of the actual model G). In the state exploration phase, we look for the chain stater state that is closest to the current state. To find such a state, we start traversing the actual model built so far in a Breadth-First manner (notice that, we are searching the graph in memory, we are not executing any event). During the traversal, we consider whether any of the unexplored transitions leads to a chain starter state. Checking if a transition leads to a chain starter is done using the parenthesis matching method explained in Section 4.4.1. This method requires to scan the bit string representation of a state having at most n bits and this takes $O(n)$ time. The number of unexplored transitions is at most $|E|$, so a traversal to look for chain starter takes $O(n \times |E|)$. Since there can be at most $|V|$ chain starter states, this traversal is done at most $|V|$ times. So, the total time spent looking for a

Function `Update(`$(v, v'; e)$`)`

Input: $(v, v'; e)$: the most recently explored transition

Output: FALSE if any hypercube assumption is violated, otherwise TRUE

$E := E \cup \{(v, v_{current}; e)\}$; `// add a new transition to the model`

$isNewState := v_{current} \notin V$; `// is this a new state?`

if *isNewState* **then**

  $V := V \cup v_{current}$;

endif

$isViolated := v_{current} \mathrel{!}= v'$; `// is this a violation?`

if *isViolated* **then**

  if *isNewState* **then**

    add to G' a hypercube based on $v_{current}$;

    $phase := stateExploration$;

  endif

  $E' = E' \setminus \{(v, v'; e)\}$; `// remove the violated, anticipated transition`

  remove from G' any vertex that has become unreachable;

endif

return $!isViolated$;

chain starter is $O(n \times |E| \times |V|)$. Once a chain starter is found, we start following the MCD chain. Finding the MCD successor of a state requires $O(n)$ time. This calculation is done at most once for each discovered state. Hence, the total time for following MCD chains is $O(n \times |V|)$.

Hence, the complexity of state exploration algorithm: $O(n \times |E| \times |V|) + O(n \times |V|) = O(n \times |E| \times |V|)$.

For the transition exploration phase, we search in the actual model for the closest unexplored transition. Again, to find such a transition, we traverse the actual model starting from the current state in a Breadth-First manner, which requires $O(|E|)$ time. This traversal is done at most $|E|$ times.

Hence, the complexity of the transition exploration algorithm: $O(|E| \times |E|) = O(|E|^2)$.

Thus, the overall complexity is $O(n \times |E|^2) + O(|E|^2) = O(n \times |E|^2)$.

The Hypercube strategy does not have a significant overhead compared to the Greedy strategy (the simple strategy of exploring the unexplored event closest to the current state) or the *optimized* implementations of the Breadth-First and the Depth-First crawling strategies (we say that an implementation of a standard crawling strategy is *optimized* if the implementation always uses the shortest known transfer sequence, as will be ex-

plained in Section 7.3). These strategies have complexity $O(|E|^2)$; the factor n is the overhead of the Hypercube strategy.

4.4.5 Proof of Optimality

In this section, we show that the Hypercube strategy is optimal for applications that are instances of the Hypercube meta-model. In particular, we show that the Hypercube strategy is optimal for both the number of resets and the number of event executions for the complete crawling of a hypercube.

In addition to the optimality of the complete crawling (exploring all transitions), we are also concerned with finding all the states in the hypercube first. It is a fact that the number of resets required to visit all the states of the hypercube is also optimal since we are using an MCD of the hypercube, which is by definition the smallest number of chains (thus of resets) to go over all the states. However, for the number of events executed to visit all the states, the Hypercube strategy is deliberately not optimal. This is because, when we come to the end of an MCD chain, we continue exploration from the current state instead of resetting and executing another MCD chain immediately, in order to keep the number of resets required for the overall crawling at the optimal value. If we do not extend the MCD chains, then this number will be optimal, but in that case the number of resets is not optimal anymore for the complete crawl.

Number of Resets

First, we consider the number of resets. Notice that, in a hypercube, each transition leaving a state at level i enters a state at level $i+1$. Once we are at a state at level i, unless we reset, there is no way to reach another state at level $j \leq i$. This means, if in total there are k transitions leaving level i, then to traverse each one at least once, we have to reset the application k times (counting the first load also as a reset). Since the level with the largest number of outgoing transitions is the middle level, the lower bound on the number of resets for crawling the hypercube is $r^* = \binom{n}{\lfloor n/2 \rfloor} \times \lceil n/2 \rceil$ where $\binom{n}{\lfloor n/2 \rfloor}$ is the number of states at the middle level and $\lceil n/2 \rceil$ is the number of outgoing transitions for a state at the middle level.

We first introduce the notation that will be used in the following. Let $C_H = \{C^1, C^2, \ldots, C^m\}$ denote the set of all chains executed by the Hypercube strategy when crawling a hypercube of dimension n. In particular $C^i = \{v_0^i, v_1^i, \ldots, v_k^i\}$ represents the i-th chain executed by the strategy where v_j^i is the state at level j. We show that the

number of chains executed by the Hypercube strategy for crawling a hypercube is r^* (i.e., $m = r^*$), so only r^* resets are used by the Hypercube strategy. We begin by the following properties of the Hypercube strategy.

Lemma 4.4.1. *Let $C^u \in C_H$ such that $u \leq r^*$, then there exists a transition $t = (v_i^u, v_{i+1}^u)$ with $i \leq \lfloor n/2 \rfloor$ such that C^u is the first chain to traverse t.*

Proof. If $u \leq \binom{n}{\lfloor n/2 \rfloor}$, then C^u is executed during the state exploration phase and contains an MCD chain $C_{MCD}^u \in C^u$. If $u = 1$, then the statement holds trivially. Let (v_i^u, v_{i+1}^u) be the transition traversed to reach the chain starter v_{i+1}^u of C_{MCD}^u. v_{i+1}^u cannot be in the upper half since a chain starter is either at the middle level or in the lower half. Hence, $i + 1 \leq \lfloor n/2 \rfloor$ implies $i < \lfloor n/2 \rfloor$ and obviously (v_i^u, v_{i+1}^u) was untraversed before C^u. If $u > \binom{n}{\lfloor n/2 \rfloor}$, then C^u is executed during the transition exploration phase. According to the transition exploration strategy, the first untraversed transition of C^u will be the one that is closest to the bottom among the untraversed transitions in the hypercube. So, unless all transitions leaving the middle level are traversed, the source of the first untraversed transition of C^u cannot be at a higher level than the middle. Since r^* chains are needed to traverse all the transitions leaving the middle level and $u \leq r^*$, there exists $t = (v_i^u, v_{i+1}^u)$ with $i \leq \lfloor n/2 \rfloor$ such that C^u is the first chain to traverse t. □

Lemma 4.4.2. *If a chain $C^u \in C_H$ enters state v_i^u using an already traversed transition and leaves v_i^u using a previously untraversed transition, then at the time C^u is executed, all transitions entering v_i^u have already been traversed.*

Proof. Let $t = (v_{i-1}^u, v_i^u)$ and $t' = (v_i^u, v_{i+1}^u)$ be the transitions that C^u traversed to enter and leave v_i^u. If $u \leq \binom{n}{\lfloor n/2 \rfloor}$, then C^u is executed during the state exploration phase and contains an MCD chain $C_{MCD}^u \in C^u$. If t and t' are the transitions on the subpath that leads to the chain starter of C_{MCD}^u, then t' is the first untraversed transition in C^u. Since the Hypercube strategy tries to use unexplored transitions as much as possible, all transitions entering v_i^u must have already been traversed in this case. Notice that, t cannot be a transition in C_{MCD}^u since all transitions in an MCD chain is traversed for the first time by the MCD chain. Also, t cannot be a transition that is used to extend C_{MCD}^u towards the top since only previously untraversed transitions are used for this purpose. If $u > \binom{n}{\lfloor n/2 \rfloor}$, then C^u is executed during the transition exploration phase. Assume that there is an untraversed transition (v_{i-1}', v_i^u). Let (v_{j-1}^u, v_j^u) be the first untraversed transition in C^u. Obviously, $i \neq j$. If $i < j$, (v_{j-1}^u, v_j^u) cannot be the first previously untraversed transition in C. The transition exploration strategy would traverse (v_{i-1}', v_i^u)

first since it is closer to the bottom of the hypercube. If $i > j$, then there is no untraversed transition leaving v^u_{i-1}, otherwise, the transition exploration strategy would prefer the untraversed one instead of t. But, then transition exploration strategy would not execute t at all since the transition exploration strategy does not continue further from the current state (v^u_{i-1}) when it has no untraversed transition. □

Lemma 4.4.3. *If $C^u \in C_H$ contains a transition $t = (v^u_i, v^u_{i+1})$ traversed for the first time by C^u such that $i \leq \lfloor n/2 \rfloor$, then C^u is also the first chain traversing every transition following t in C^u and the size of C^u is at least $\lfloor n/2 \rfloor + 1$.*

Proof. As explained in Section 4.4.1, in a chain executed by the Hypercube strategy, all transitions following the first unexplored transition are also unexplored. So, C^u is the first chain traversing all transitions after t in C^u.

Now, we show that the size of C^u is at least $\lfloor n/2 \rfloor + 1$. The case $i = \lfloor n/2 \rfloor$ is trivial. Assume $i < \lfloor n/2 \rfloor$. If C^u is executed during the state exploration phase, then there is an MCD chain $C^u_{MCD} \in C^u$. Since every MCD chain contains a state in the middle level, C^u_{MCD} also contains a state in the middle level, namely $v^u_{\lfloor n/2 \rfloor}$. If $v^u_{\lfloor n/2 \rfloor}$ has a successor $v^u_{\lfloor n/2 \rfloor+1}$ in C^u_{MCD}, we will definitely reach $v^u_{\lfloor n/2 \rfloor+1}$. Otherwise, since it is the first time $v^u_{\lfloor n/2 \rfloor}$ is visited, none of its outgoing transitions are traversed before and the strategy will follow one of them. Assume C^u is executed during the transition exploration phase. Let (v^u_j, v^u_{j+1}) be any transition traversed by C^u such that $j \geq i$. We know that (v^u_j, v^u_{j+1}) is untraversed before C^u. By Lemma 4.4.2, if a transition t' entering v^u_{j+1} was already traversed at the time C^u is executed, then among all chains that traversed t', only the one that traversed t' for the first time left v^u_{j+1} using a previously untraversed transition. (If at all, others used an already traversed transition to leave v^u_{j+1}). Combining this with the fact that in a hypercube, for all levels $j \leq \lfloor n/2 \rfloor$, the number of transitions leaving a state v_j at level j is greater than or equal to the number of transitions entering v_j, we can conclude that there is an untraversed transition leaving v^u_{j+1} and the transition exploration strategy will follow it. □

Lemma 4.4.4. *Each transition leaving the middle level is traversed by the first r^* chains.*

Proof. This is a consequence of Lemma 4.4.1 and Lemma 4.4.3. □

Lemma 4.4.5. *Each transition leaving a state in the lower half of the hypercube is traversed by the first r^* chains.*

Proof. Assume there exists a transition leaving a state in the lower half of the hypercube that has not been traversed after r^* chains. Let v_i be a state at level i such that $i < \lfloor n/2 \rfloor$

and (v_i, v_{i+1}) is untraversed after r^* chains. Let $C = \{v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_k\}$ be the chain that traverses (v_i, v_{i+1}) for the first time. By Lemma 4.4.3, $k > \lfloor n/2 \rfloor$ and all transitions (v_j, v_{j+1}) in C where $i \leq j \leq k$ are also traversed for the first time by C. In particular, $(v_{\lfloor n/2 \rfloor}, v_{\lfloor n/2 \rfloor+1})$ was untraversed before C. But, this contradicts Lemma 4.4.4. □

Lemma 4.4.6. *Each transition entering a state in the upper half of the hypercube is traversed by the first r^* chains.*

Proof. In a hypercube, the number of transitions entering a state in the upper half is greater than the number of transitions leaving it. So, each state in the upper half must be visited more than the number of its outgoing transitions. The Hypercube strategy continues to traverse untraversed transitions as long as the current state has one. That means, if all incoming transitions of an upper level state are traversed, then all outgoing transitions are also traversed by the strategy. By Lemma 4.4.4, every transition entering the level $\lfloor n/2 \rfloor + 1$ (first level of the upper half) is traversed by r^* chains. The result extends similarly for the higher levels. □

Theorem 4.4.7. *Every transition in the hypercube is traversed by the first r^* chains of the Hypercube strategy.*

Proof. Follows from Lemma 4.4.5 and Lemma 4.4.6. □

Number of Event Executions

The Hypercube strategy is also optimal in terms of the total number of events executed to crawl an application whose model is a hypercube. The optimal number of event executions required to crawl a hypercube of dimension $n \geq 2$ is $e^* = 2^{n-2}n + r^*n/2$. For simplicity, assume n is even, then this formula can be derived as follows (the number is also valid for odd n). As we have shown, at least r^* chains are needed to crawl the hypercube and each of these chains has to cover a transition leaving a middle level state. As a result, some transitions in the lower half must be traversed more than once. Since there are r^* chains to execute, for $1 \leq i \leq n/2$, the total number of transition traversals for transitions entering level i is r^*. So, for all transitions whose source is in the lower half, $r^* \times n/2$ event executions are needed. Each transition whose source is in the middle level or in the upper half needs to be executed at least once. There are $2^{n-2} \times n$ such transitions. So, in total, the optimal number of event executions to crawl the hypercube

is $e^* = 2^{n-2} \times n + r^* \times n/2$. It is not difficult to see that the Hypercube strategy uses the optimal number of event executions as stated by the following.

Theorem 4.4.8. *The Hypercube strategy executes exactly e^* events when crawling a hypercube application.*

Proof. We have already shown that the Hypercube strategy uses the optimal number of chains which is r^*. So, in the lower half of the hypercube no more than $r^* \times n/2$ events will be executed. Then, it is enough to show that each transition leaving a state at level $\lfloor n/2 \rfloor \leq i \leq n$ is traversed exactly once. By Lemma 4.4.1 and Lemma 4.4.3, each chain $C^u \in C_H$ is the first chain to traverse some $t = (v_i^u, v_{i+1}^u)$ with $i \leq \lfloor n/2 \rfloor$ and every transition after t is also traversed for the first time by C^u. That includes all transitions traversed by the Hypercube strategy in the upper half. Hence, there is no chain that re-traverses a transition in the upper half. Since we have also shown that the Hypercube strategy covers all transitions, each transition in the upper half is traversed exactly once. □

4.5 Conclusion

Model-based crawling is an approach for designing efficient crawling strategies for RIAs. In this approach, we aim at discovering the states of the application as early as possible by trying to anticipate the model of the application. The Hypercube strategy is the first strategy following this approach. The Hypercube strategy is based on the Hypercube meta-model, and it is an optimal crawling strategy for this model. The new version of the Hypercube strategy that is presented in this thesis overcomes the practical shortcomings of the initial version which was introduced earlier.

As we present in Chapter 7, our experimental results show that the Hypercube strategy is significantly more efficient than the Breadth-First and Depth-First strategies. This is true even when the application that is being crawling does not conform to the Hypercube meta-model at all. However, one difficulty with the Hypercube meta-model is that its assumptions are often too strict to be valid in most RIAs, so it is difficult to find real examples where the full strength of the strategy is employed. As we explain in the next chapter, the strategies that were designed after the Hypercube strategy are less strict in the sense that they observe the behavior of the events in the application before making any commitment to an assumption and perform even better.

Chapter 5

The Probability Strategy

5.1 Introduction

The second model-based strategy is based on what has been called the *Menu* meta-model. The Menu meta-model and the corresponding crawling strategy have been introduced in [26] (in Suryakant Choudhary's master thesis). Compared with the Hypercube meta-model, the Menu meta-model has more realistic assumptions, thus the Menu strategy performs better than the Hypercube strategy in most cases.

The discussions during the design of the Menu strategy led to a new strategy: the *Probability* strategy, which is explained in this chapter. The Probability strategy is based on a statistical model. Compared with the Hypercube and the Menu, the Probability strategy is very relaxed in terms of assumptions. It is much simpler than both in terms of implementation and comparable to the Menu strategy in terms of strategy efficiency.

In this chapter, we start with an overview of the Menu strategy in Section 5.2. The overview of the Probability strategy is presented in Section 5.3. The formulation used by the Probability strategy to prioritize events is given in Section 5.4. The algorithm which uses this formulation to choose the next event to explore is explained in Section 5.5. We introduce some alternative versions of the Probability strategy in Section 5.6 and conclude the chapter in Section 5.7.

5.2 Overview of the Menu Strategy

The Menu strategy is based on the Menu meta-model. The Menu meta-model is formed on the assumption that the result of an event execution is independent of the state

(source state) where the event has been executed. That is, all instances of an event are expected to lead to the same state. This captures the types of events that are used for navigation in an application such that whenever they are executed they lead to the same page (for example, a menu may contain "home", "contact", "about" etc.).

The Menu strategy works by categorizing each event found in the application. There are three different categories and an event is classified after two instances of the event have been explored (i.e., after the event has been executed from two different states). These categories are

- *Menu Events:* A menu event is an event that resulted in the same state in its first two explorations.
- *Self-Loop Events:* A self-loop event is an event that did not cause any state change in its first two explorations.
- *Other Events:* All the remaining events which had neither of the above behaviors in their first two explorations.

Initially, each event is assumed to behave like a menu event, even though it is not categorized yet. That means, in its second exploration an event is expected to lead to the same state that was reached in its first exploration. Based on the result of the second exploration, an event is categorized and its category is never changed afterward even if some of its instances violate the expected behavior of its category later on.

During the state exploration phase, the Menu strategy aims to explore uncategorized events (i.e., the events that have not been explored at all from any state or explored just once) and the events that are categorized as "other". Among these events, the priority is given to the events that have not been explored at all. The events categorized as menu or self-loop are not explored explicitly during this phase since they are not expected to result in a new state. However, an unexplored menu event can still be used to build a transfer sequence since the result of an unexplored menu event can be predicted. Of course, in such cases, after the menu event is executed in the transfer sequence, it is checked whether the reached state is the one predicted. If the reached state is not the predicted one, the strategy makes a new decision regarding what to do next from the reached state.

Once all the instances of uncategorized and "other" events have been explored, the strategy moves on to the transition exploration phase to explore the remaining menu and self-loop events. As an optimal transition exploration strategy, the Menu strategy

requires to solve the Rural Chinese Postman Problem (RCPP), which is the problem of finding a least cost path that traverses a subset of all edges in a known graph. Since the RCPP is an NP-Hard problem [34], the Menu strategy, instead, uses a solution to an easier problem, the Chinese Postman Problem (CPP), which is the problem of finding a least cost path that traverses all the edges in a known graph.

Unlike the Hypercube strategy, which makes strict assumptions about an event's behavior without any observation of the actual behavior of the event, the Menu strategy observes two explorations of an event before finalizing the assumption about the event's behavior. The Menu strategy then uses these observations to differentiate between the events that are less likely to lead to new states (menu and self-loop events) and the ones which are more likely ("other events"), so that the latter are given priority. This idea of observing the events' behavior and prioritizing the events based on their likelihood of discovering a new state is the basis for a new strategy that we discuss next.

5.3 Overview of the Probability Strategy

In a RIA, each event can be associated with the set of states that are reached by exploring the instances of the event. This set can be referred to as the "destination set" of the event. The Menu strategy works by separating the events that are likely to have singleton destination sets (the "menu" events) from the events that are likely to have non-singleton destination sets, so that the latter are given exploration priority after the self-loop events are excluded. However, the Menu strategy relies on very few observations for categorization and does not present any further measure to compare two events that are in the same the category.

With the Probability strategy, we attempt to take into account the size of the destination set of an event. On one extreme of the range, there are menu events (having a singleton destination set), and on the other extreme, there are events that lead to a different state from each source state. The destination set of an event can be anywhere on this range. It is possible to estimate where the destination set of an event is more likely to be located on this range by looking at the results observed during previous executions of the event. However, using such an estimation is not a good choice for prioritization since the possible overlaps between the destination sets of the events are not taken into account. It is not reasonable to give priority to an event simply because it leads to many different states which have already been reached by other events. Since the goal is to discover states, it is more reasonable to look at how often an event discovers a state (i.e.,

leads to a state that has not been reached before by any event).

The Probability strategy is a strategy based on the hypothesis that an event which was often observed to lead to new states in the past is more likely to lead to new states in the future. In this strategy, an event's probability of discovering a new state on its next exploration is estimated based on how successful it was in its past explorations. The formulation used for this purpose is explained in Section 5.4.

The Probability strategy aims at choosing the action that maximizes the probability of discovering a state. This choice is not solely based on a comparison of the "raw" probabilities of the events. The length of the transfer sequence that needs to be executed in order to explore an event is also taken into account. In Section 5.5, we first introduce a mechanism to compare a pair of events and then give an algorithm which uses this mechanism to choose the action that maximizes the probability of discovering a state.

5.4 Estimating an Event's Probability

5.4.1 Rule of Succession

The problem of estimating an event's probability of discovering a new state, given the observations of its previous explorations, is analogous to the following mathematical example, that is known as the "(finite[1]) rule of succession" [68]

Suppose there is an urn with a finite but unknown number of balls, each of which is either red or white. m balls are drawn at random without replacement from the urn such that p of the drawn balls are observed to be white. If nothing is known about the relative proportion of red and white balls in the urn, what is the probability that next ball drawn will be white?

When it is assumed that any proportion of red and white balls is equally likely, the answer to this question is given by the Bayesian formula: $\frac{p+1}{m+2}$ [68].

5.4.2 Probability of an Event

Using the rule of succession, we can estimate the probability, $P(e)$, of discovering a new state on an event e's next exploration:

$$P(e) = \frac{S(e) + p_s}{N(e) + p_n} \tag{5.1}$$

[1]The original formulation of the rule of succession by Laplace supposes that the trials can be repeated infinitely many times, but the same result is still applicable to the finite case [68]

- $N(e)$ is the "exploration count" of e, that is, the number of times e has been explored (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 $N(e)$ explorations,
- p_s and p_n are called the pseudo-counts and they represent the "initial success count"' and the "initial exploration count", respectively. These terms are preset and define the prior (initial) probability for an event.

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 (notice that, $N(e) = S(e) = 0$ initially). To minimize the effect of the pseudo-counts on the probability, one may prefer to use smaller (possibly non-integer) numbers, such as $p_s = 0.5$ and $p_n = 1$ for an initial probability of 0.5.

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 having events that get a probability of 0 because no new state were found at their first exploration. With our formula, events never have a probability of 0 (or 1) and can always be picked up after a while.

After each exploration of an event, the event's probability is updated. As we explore an event and have more observations about its behavior, the weight of the initial probability decreases and actual observations dominate the value of the probability.

5.5 Choosing the Next Event to Explore

Once we know how to estimate the probability of an event, the next step is to design an algorithm that uses these probabilities to choose the event to explore next. Knowing that among the unexplored events in a given state the best choice is the one with the highest probability, this problem is better expressed as choosing a state where the next event should be explored. In this section, we provide an algorithm to choose a state that maximizes the probability of discovering a state. In the remainder, the following notation is 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.

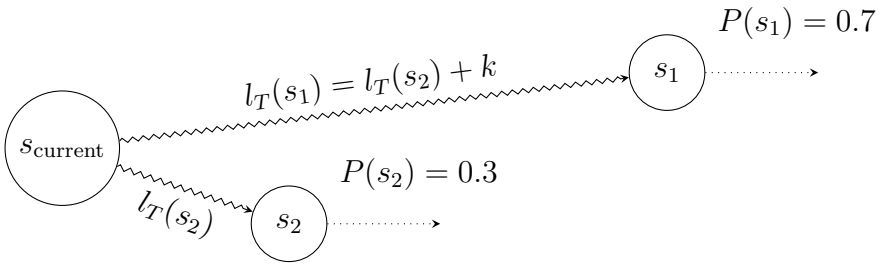

Figure 5.1: Two candidate states where it is costlier to reach the one with the higher probability.

- s_{current} represents the current state, the state we are currently at in the application.
- For a state $s \in S$, we define the probability of the state, $P(s)$, as the probability of the event that has the maximum probability among all the unexplored events of s. If s has no unexplored events, then $P(s) = 0$.
- $l_T(s)$ is the length of the shortest known transfer sequence from s_{current} to s.

A simple algorithm would be choosing the state which has the highest probability. However, this might not always be the best choice[2]. For instance, consider the case depicted in Figure 5.1 where we have two states s_1 and s_2 with unexplored events such that $P(s_1) > P(s_2)$ and $l_T(s_1) > l_T(s_2)$. In this case, it is not fair to choose s_1 just because $P(s_1) > P(s_2)$. This ignores the fact that by choosing s_1 we are executing $k = l_T(s_1) - l_T(s_2)$ more events as part of the transfer sequence. For a fairer comparison, we should consider the probabilities when an equal number of events are executed.

k-Iterated Probability of a State To compare a pair of states s_1 and s_2, in the case $P(s_1) > P(s_2)$ and $l_T(s_1) > l_T(s_2)$ (note that this is the only case where making a decision is not trivial), we consider the difference of the lengths of the transfer sequences, k, and calculate the "k-iterated probability" of s_2 (the state that is closer to the current state), noted as $P(s_2, k)$.

We define $P(s_2, k)$ as the probability of discovering a new state by choosing s_2 and exploring k more events after the event in s_2 is explored. Thus, $P(s_1)$ and $P(s_2, k)$,

[2]The efficiency comparison of this simple algorithm with the proposed algorithms can be found in Section A.2 of Appendix A. The results show that using the proposed algorithm discovers the majority of the states faster than this simple algorithm.

which are the probabilities of finding a new state when an equal number of events are executed, can be used to compare s_1 and s_2.

The difficulty in calculating the iterated probability $P(s,k)$ for a state s is that when we explore an event from s, we do not know which state is going to be reached; therefore, we do not know the probability of this reached state. To address this problem, we estimate the probability, P_{avg}, of exploring an event from any state.

Assuming that reaching any state is equally likely, we define P_{avg} as the average probability of all the states:

$$P_{\text{avg}} = \frac{\sum_{s \in S} P(s)}{|S|} \tag{5.2}$$

With P_{avg}, it is easy to compute $P(s,k)$ using $(1 - P(s))(1 - P_{\text{avg}})^k$, the probability of not discovering a state by choosing s and then not discovering any state by exploring k more events (each with probability P_{avg}) afterwards:

$$P(s,k) = 1 - (1 - P(s))(1 - P_{\text{avg}})^k \tag{5.3}$$

5.5.1 Algorithm

Based on the discussion above, we present Algorithm 7 to decide on a state, s_{chosen}, where the next event should be explored. The algorithm chooses the state that maximizes the probability of discovering a state. In other words, a state $s \in S$ that satisfies the following condition becomes s_{chosen}.

$$\begin{aligned}
&\forall s' \in S \\
&\text{if } l_T(s) > l_T(s'), \quad P(s) \geq P(s', l_T(s) - l_T(s')) \\
&\text{if } l_T(s) \leq l_T(s'), \quad P(s, l_T(s') - l_T(s)) \geq P(s')
\end{aligned}$$

The first case makes sure that choosing s has a better probability than choosing any s' which is closer to the current state than s is. The second case makes sure that choosing s has a better probability than any other state s' that is not closer to the current state than s is (notice that, when $l_T(s) = l_T(s')$, we compare directly $P(s)$ and $P(s')$, since $P(s, l_T(s') - l_T(s)) = P(s,0) = P(s)$).

The algorithm initializes the variable s_{chosen} to the current state and searches for a better state in iterations. At iteration i, s_{chosen} is compared with the state s that has the highest probability among the states that are i event executions away from the current state (line 6). That is, the iterated probability of s_{chosen} compared with $P(s)$ to decide

if s is preferable to s_{chosen} (i.e., $P(s) > P(s_{\text{chosen}}, l_T(s) - l_T(s_{\text{chosen}}))$). If this is the case, then s becomes the new s_{chosen}.

This search continues until it is impossible to find a state which could be a better choice than s_{chosen}. It is possible to calculate how many iterations later we can be sure that s_{chosen} is the best choice. This is because, after each iteration (that we do not find a better state), the iterated probability of s_{chosen} increases and at some point, the iterated probability will be greater than the probability, P_{best}, of the state that has the maximum probability among all states. When the iterated probability reaches that point, it is not required to search any further since even a state with the highest probability is not preferable to s_{chosen} anymore. To calculate the number of iterations we should keep searching for a better state, the function "maxDepthToCheckAfter" is used as explained below.

Algorithm 7: ChoosingTheNextState

Require: There is an unexplored event
Input: The extracted model of the application so far
Input: s_{current} the current state
Output: a state s_{chosen} where next event should be explored

```
1  s_chosen := s_current ;
2  i := 0 ;
3  depthToCheck := maxDepthToCheckAfter(s_chosen);
4  while i < depthToCheck do
5      Let s be the state with max probability among all s_i with l_T(s_i) = i;
6      if s is preferable to s_chosen then
7          s_chosen := s ;
8          depthToCheck := i + maxDepthToCheckAfter(s_chosen) ;
9      i := i + 1 ;
10 return s_chosen ;
```

Calculating maxDepthToCheckAfter: Whenever we set s_{chosen}, it is possible to calculate a value d such that we can stop the search in case we do not see a state that is preferable to s_{chosen} in d more iterations. This value is the smallest (integer) d that satisfies the following inequality

$$P(s_{\text{chosen}}, d) \geq P_{\text{best}} \tag{5.4}$$

where P_{best} is the probability of a state with maximum probability (we just need the value of P_{best}, we do not need the location of a state with probability P_{best} in the model). By solving this inequality for d, we define maxDepthToCheckAfter[3]:

$$\text{maxDepthToCheckAfter}(s_{\text{chosen}}) = \begin{cases} \lfloor \log_{(1-P_{\text{avg}})}(\frac{1-P_{\text{best}}}{1-P(s_{\text{chosen}})}) \rfloor & \text{if } P(s_{\text{chosen}}) > 0 \\ \infty & \text{if } P(s_{\text{chosen}}) = 0 \end{cases} \quad (5.5)$$

We consider this depth to be infinity in the case $P(s_{\text{chosen}}) = 0$. That means, when we initialize s_{chosen} to the current state (line 1 of the algorithm), if there is not any unexplored event in the current state (i.e., $P(s_{\text{chosen}}) = 0$), then we cannot set a depth bound on the search until a state with an unexplored event is found. To prevent an infinite loop, the existence of a state with an unexplored event is required as a precondition of the algorithm.

We remind again that if a better state is found within this calculated depth, then this calculation is done again for the new s_{chosen}.

5.5.2 Complexity Analysis

Considering the extracted model of the application as a graph $G = (V, E)$, the worst-case time complexity of the Probability strategy can be expressed in terms of the size of the model, $|E|$, and the maximum number of events in a state, noted as n (i.e., n is the maximum outdegree of G). In this analysis, we assume that the unexplored events of each state are maintained in a binary heap data structure using the probabilities of the events as keys. The complexity of the operations required to decide on an event and exploring the event can be analyzed as follows.

- Finding the unexplored event with the maximum probability in a state is an $O(1)$ operation (i.e., the operation find max in a heap).

[3] The first case of the formula 5.5 is obtained by solving the inequality 5.4 for d as follows.

$$\begin{aligned} P(s_{\text{chosen}}, d) &\geq P_{\text{best}} \\ 1 - (1 - P(s_{\text{chosen}}))(1 - P_{\text{avg}})^d &\geq P_{\text{best}} \\ 1 - P_{\text{best}} &\geq (1 - P(s_{\text{chosen}}))(1 - P_{\text{avg}})^d \\ \tfrac{1-P_{\text{best}}}{(1-P(s_{\text{chosen}}))} &\geq (1 - P_{\text{avg}})^d \end{aligned}$$

Taking the log of both sides to base $(1 - P_{\text{avg}})$, the smallest integer d that satisfies this inequality is found:

$$\lfloor \log_{(1-P_{\text{avg}})}(\frac{1 - P_{\text{best}}}{1 - P(s_{\text{chosen}})}) \rfloor$$

- Calculating P_{avg} and P_{best} requires $O(|V|)$ time.
- Traversing the current model to determine s_{chosen} requires $O(|E|)$ time.
- Updating an event's probability after the event is explored requires to go over each state that the event is still unexplored and do a decrease or increase key operation on the heap that maintains the unexplored events of the state. This takes $O(|V| \log n)$ time.
- Removing the explored event from the heap of the s_{chosen} takes $O(\log n)$ time (i.e., delete max operation).

So, the total time for a single event exploration is $O(1)+O(|V|)+O(|E|)+O(|V| \log n)+O(\log n) = O(|E| + |V| \log n)$

Since there are $|E|$ events to explore, the total time for exploring all the events is $O(|E|^2 + |E||V| \log n)$. The term $|E||V| \log n$ is the overhead of the Probability strategy when compared with the Greedy and the optimized versions of Depth-First and Breadth-First which have a complexity of $O(|E|^2)$.

5.6 Alternative Versions of the Strategy

We consider the described algorithm so far as the *default* version of the Probability strategy and use this default version when comparing the probability strategy with other strategies in the experiments.

We have also experimented with several alternative versions of this default strategy. In this section, we explain these alternative versions. However, with these alternative versions, there were no overall significant improvement on the results across all the applications that were used in the experiments. Although it is sometimes the case that an alternative version yield better results than the default version, the improvement is either not very significant or specific to some of the applications. The results for these alternative versions are provided in Appendix A.

Alternative Algorithm to Choose a State

When deciding on the state where the next event should be explored, the default strategy chooses the state that maximizes the probability of discovering a state. An alternative algorithm is to choose the state which minimizes the expected cost (number of event

executions) to discover a state. The two methods are not so different, except that the alternative method considers the cost of discovering a state more explicitly than the default one.

For a state s, the expected cost of discovering a state by choosing s, noted as $E(s)$, is calculated by using P_{avg} as the probability of discovering a state in each event exploration subsequent to the event in s:

$$E(s) = P(s)(l_T(s) + 1) + \sum_{k=1}^{\infty}(1 - P(s))(1 - P_{\text{avg}})^{k-1}P_{\text{avg}}(l_T(s) + k + 1)) \tag{5.6}$$

The first term $P(s)(l_T(s) + 1)$ is the expected cost of discovering a state by exploring the event in s where $l_T(s) + 1$ is the cost of exploring an event from s. The summation is for the subsequent explorations that would be done in case a state is not discovered in the previous explorations: $(1 - P(s))(1 - P_{\text{avg}})^{k-1}P_{\text{avg}}$ is the probability of discovering a state in the k-th subsequent exploration and $(l_T(s) + k + 1)$ is the associated cost.

The infinite sum in $E(s)$ converges to the following value[4]:

$$(1 - P(s))(l_T(s) + 2 + \frac{1 - P_{\text{avg}}}{P_{\text{avg}}}) \tag{5.7}$$

Thus, the formula to compute $E(s)$:

$$E(s) = P(s)(l_T(s) + 1) + (1 - P(s))(l_T(s) + 2 + \frac{1 - P_{\text{avg}}}{P_{\text{avg}}}) \tag{5.8}$$

For this alternative version, the algorithmic framework given in Algorithm 7 still applies. In this case, the state that has a smaller expected cost is preferable when comparing two states. In addition, it is still possible to calculate for how many more iterations the search for a better state should continue, as we explain next.

[4] The value of the infinite summation is obtained as follows. We first rewrite the summation by placing the free terms outside:

$$(1 - P(s))P_{\text{avg}}\sum_{k=1}^{\infty}(1 - P_{\text{avg}})^{k-1}(l_T(s) + k + 1))$$

The terms in the new summation form an arithmetico-geometric sequence [63] whose n-th term is defined to be $[a + (n - 1)d]r^{n-1}$. In our case, we have $a = l_T(s) + 2$, $d = 1$ and $r = 1 - P_{\text{avg}}$. When $-1 < r < 1$, the infinite arithmetico-geometric series converges to

$$\frac{a}{1 - r} + \frac{dr}{(1 - r)^2}$$

Applying this in our case, we obtain the value in 5.7

Calculating maxDepthToCheckAfter: The maximum depth to check after s_{chosen} is set is the smallest (integer) d that satisfies the following inequality

$$E(s_{\text{chosen}}) \leq (d + l_T(s_{\text{chosen}}) + 1)P_{\text{best}} + (1 - P_{\text{best}})(d + l_T(s_{\text{chosen}}) + 2 + \frac{1 - P_{\text{avg}}}{P_{\text{avg}}}) \quad (5.9)$$

The right hand side of the inequality is the expected cost of a state that is $d + l_T(s_{\text{chosen}})$ event executions away from the current state and that has the maximum probability, P_{best}.

Solving this inequality for d, the function maxDepthToCheckAfter is defined:

$$\text{maxDepthToCheckAfter}(s_{\text{chosen}}) = \begin{cases} \lfloor E(s_{\text{chosen}}) - \frac{1-P_{\text{avg}}-P_{\text{best}}}{P_{\text{avg}}} - l_T(s_{\text{chosen}}) - 2 \rfloor & \text{if } P(s_{\text{chosen}}) > 0 \\ \infty & \text{if } P(s_{\text{chosen}}) = 0 \end{cases} \quad (5.10)$$

As in the default version, if the current state does not have an unexplored event (initially, $P(s_{\text{chosen}}) = 0$), we do not set a depth bound until a state with an unexplored event is encountered during the search.

Alternative Probability Estimation

An alternative way to estimate the probability of an event is to give more weight to the results of the event's recent explorations. This allows the strategy to react more quickly to changes in an event's behavior than the default probability formula. In the default formula, the result of an exploration that happened in an early period of the crawl still has its effect on the probability even after many explorations.

We experiment with moving average techniques to give more weight to recent explorations. By doing this, we prioritize the events based on smaller observation periods. Two alternative formulations that are used for this purpose are explained below.

1. **Simple Moving Average (SMA):** This technique uses the unweighted average of the most recent w explorations of an event to calculate the event's probability. The parameter w is the window size. The formula for SMA is

$$P_{\text{SMA}}(e) = \frac{S(e, w) + p_s}{\min(N(e), w) + p_n} \quad (5.11)$$

   - $S(e, w)$ is the "success count" of e in its most recent $min(N(e), w)$ explorations

- $N(e)$, p_s and p_n are, as previously, the number of explorations of e so far, and the pseudo-counts for the initial success count and initial exploration count, respectively.

Initially, when the event has not yet been explored w times, the probability 5.11 is identical to the probability 5.1 (used for the default version). The moving average takes effect once the event is explored w times. Thus, the probability 5.1 is a special case of the probability 5.11 for $w = \infty$.

2. **Exponentially Weighted Moving Average (EWMA):** This technique calculates the probability of an event using a weighted average of the recent exploration results of the event.

   Let $\{X_n\} = X_1, \ldots, X_n$ be a sequence of binary outcomes representing the exploration history of an event, that has been explored n times so far, such that $X_i = 0$ iff in its i-th exploration, the event failed to discover a new state ($X_i = 1$, otherwise). Then, the EWMA of the sequence $\{X_n\}$, written as EWMA($\{X_n\}$), can be calculated using the following recursive formula:

$$\text{EWMA}(\{X_n\}) = \alpha X_n + (1 - \alpha)\text{EWMA}(\{X_{n-1}\}) \tag{5.12}$$

   In EWMA, the weights decrease exponentially towards the older explorations. The smoothing parameter, α, adjusts the relative importance of the result of a recent exploration. As α gets higher, the weight assigned to the result of a recent exploration increases.

   A consideration when using EWMA is to define an initial value, EWMA($\{X_0\}$), for the case when the event has not been explored yet. The smaller the value chosen as α, the longer will be the effect of the initial value on the resulting average. For this reason, we use the following formulation where the EWMA is not applied until the event explored at least $\frac{1}{\alpha}$ times.

$$P_{\text{EWMA}}(e, n+1) = \begin{cases} P(e) & \text{if } n < \frac{1}{\alpha} \\ \alpha X_n + (1 - \alpha)P_{\text{EWMA}}(e, n) & \text{if } n \geq \frac{1}{\alpha} \end{cases} \tag{5.13}$$

   where $P_{\text{EWMA}}(e, n)$ is the probability of discovering a new state on the n-th exploration of e. Note that, the default probability (formula 5.1) is used until the event executed $\frac{1}{\alpha}$ times. (For example, for $\alpha = 0.01$, we wait for 100 explorations before

applying the EWMA.) After that point on, the EWMA is used taking the current probability of the event as the initial value for the moving average.

Notice that, the formula 5.1 is the special case of the 5.13 for $\alpha = 0$.

Aging

We apply the notion of aging to prevent an event from not being picked for exploration for a long time. We define the age of an event e, Age(e), as a value between 0 and 1 using the following formula

$$\text{Age}(e) = 1 - \frac{\text{number of discovered transitions when } e \text{ was last explored}}{\text{number of transitions discovered so far}} \tag{5.14}$$

Initially, the age of an event is defined as 1. According to this formula, if e is the most recently explored event, then its age is 0. While an event is not picked for exploration, its age increases.

A threshold parameter τ is introduced such that when an event's age becomes greater than τ, then its probability is boosted (by temporarily setting its probability to P_{best}) until it is explored once again. Once an event whose probability has been boosted is picked for exploration, it continues to use its regular probability.

Alternative Estimation of P_{avg}

An alternative way to estimate P_{avg} is using the EWMA of the event explorations so far. That is, if $\{Y_n\} = Y_1, \ldots, Y_n$ is the sequence of binary outcomes for the event explorations of the crawl so far such that $Y_i = 0$ iff the i-th event exploration failed to discover a new state ($Y_i = 1$, otherwise). (Notice that, $\{Y_n\}$ is not a sequence for a specific event, it contains the result of any exploration). Then, we define

$$P_{\text{avg}} = \text{EWMA}(\{Y_n\}) \tag{5.15}$$

where EWMA($\{Y_n\}$ is as defined by the formula 5.12 with the initial value equal to the initial probability that is used for the events (i.e., EWMA($\{Y_0\}$) = p_s/p_n).

The difference between this alternative estimation and the default one is that the default estimation assumes that when we explore an event, we may end up in any one of the known states (ending up in each state is equally likely) and thus the probability of the next event we are going to explore would be the average of the probabilities of the states. In the alternative version, the expectation is that the next event we are going to explore will have a similar result as the recent explorations regardless of the state we end up.

5.7 Conclusion

We have explained the Probability strategy that uses the statistical data accumulated during the crawl to choose an event to explore such that the chosen event gives the highest probability of discovering a new state. We have first explained the default version of the strategy and then introduced some alternative versions. Although some of these alternative versions yield better results in some cases, we do not see an overall improvement in general. For this reason, we use the default version of the strategy for comparison with the other crawling strategies in Chapter 7. We present the comparison of the alternative versions with the default version in Appendix A.

The Probability strategy is a better alternative to the previous model-based crawling strategies, the Hypercube and the Menu. Experimental results show that the Probability strategy performs even better than the Hypercube strategy which is already more efficient than the Breadth-First and Depth-First. The Probability and the Menu often have comparable results in terms of number of events executed and the number of resets. An advantage of the Probability strategy over the Menu is its simpler implementation since the Menu strategy solves a Chinese Postman Problem (CPP) for its transition exploration strategy. Although CPP is solvable in polynomial time, efficiently implementing a solver for CPP is not so straight-forward [65].

Chapter 6

Crawler Implementation

6.1 Introduction

In this chapter, we explain the implementation details of our crawler. Our crawler has been implemented as a prototype of IBM® Security AppScan® [44]. AppScan is an automated web scanner which aims at detecting security vulnerabilities and accessibility issues in web applications. AppScan originally does not have any crawling strategy for RIAs.

Some of the existing components in AppScan are used to build our crawler. These are the DOM Equivalence algorithm and the embedded browser (which includes the functionalities of JavaScript execution and DOM manipulation). Except for the Greedy and the Menu strategies, I implemented all the mentioned crawling strategies. For event identification, the functionality of detecting the elements that have registered events is provided by AppScan, but I implemented the algorithms that produce event identifiers.

We explain the crawler architecture in Section 6.2. The details of event identification algorithms are presented in Section 6.3. This is followed by the DOM Equivalence algorithm in Section 6.4. In Section 6.5, we conclude the chapter.

6.2 Crawler Architecture

Figure 6.1 shows the simplified RIA crawler architecture. The AppScan component performs the functionalities of a browser. It is able to construct the initial DOM for a given URL, execute JavaScript using its *JavaScript Engine* sub-component and perform the required manipulations to update the current DOM. It also contains the sub-components

for *Event Identification* and *DOM Equivalence*. The Event Identification component detects the DOM elements that have enabled events on the current DOM and produces identifiers for these events. The DOM Equivalence component generates the DOM identifier for the current DOM. This information is used by the crawling strategies. The crawling strategies are implemented as a separate module that can be called from AppScan.

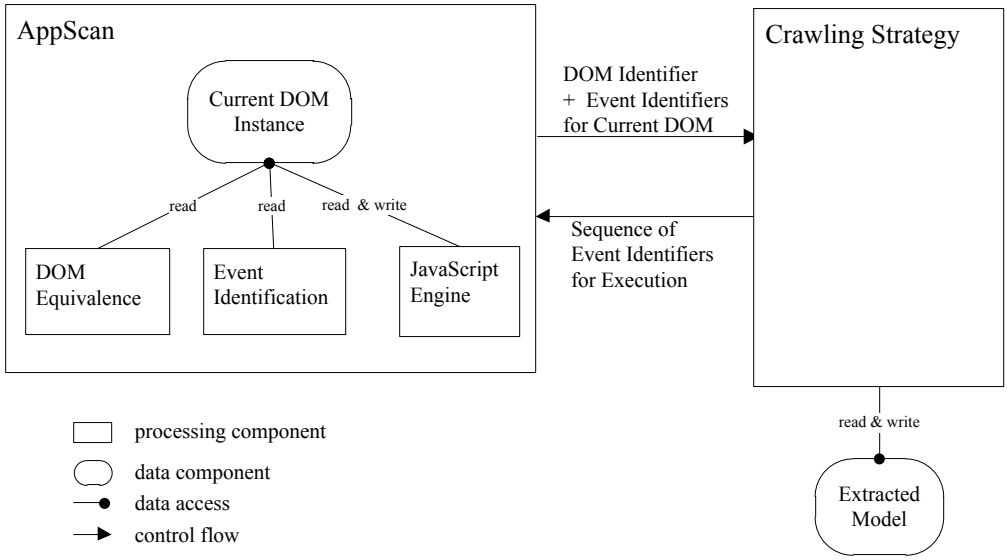

Figure 6.1: RIA Crawler Architecture

When the AppScan component is given a URL to crawl, it retrieves the contents from the server, and constructs the DOM. Later, it generates the set of event identifiers for the events found in the DOM and then generates the identifier for the DOM. (Details of how event and DOM identifiers are generated are explained below). The DOM identifier and the list of event identifiers are passed to the crawling strategy. The crawling strategy then decides on an event to explore and returns an event sequence. The event sequence consists of the event to explore appended to a (possibly empty) transfer sequence, which will take the crawler to the DOM from where the event will be explored. The AppScan component executes the received sequence of events and returns the control back to the crawling strategy, providing the event identifiers and the DOM identifier for the reached DOM. Whenever the crawling strategy takes control, it updates the extracted model according to the result of the last event exploration. This process continues until there is no unexplored event left and thus a model for the URL is extracted.

6.3 DOM Events and Event Identification

An important component for a RIA crawler is the algorithm to identify the events in a DOM. An event identifier is used to differentiate between the user actions that can be exercised in the application. It allows the crawler to recognize an event in a state when the state is visited at different times during the crawl, so that the crawler can keep track of unexplored events in the state or trigger explored transitions from the state. In addition, event identifiers are used for detecting common events in different states. This allows crawling strategies to make predictions about an event's behavior.

To compute an event identifier, the following information is needed:

- **DOM Element Identifier:** Since the events are associated with DOM elements, it is necessary to compute an identifier for the DOM element to which the event is registered. A DOM element identifier is used to differentiate an element from the other elements in the same DOM, as well as to recognize the element when it is seen in a different DOM or when an equivalent DOM is visited again later.

- **Event Type:** This is the name of the user action (onmouseover, onclick, etc.). This information is needed since the same DOM element can react to multiple types of events.

- **Event Handler:** This is the JavaScript code that will be executed when the event is triggered. Event handler defines what happens when the event occurs. As we explain below, it is possible to change the event handlers that are registered to a DOM element through JavaScript execution. That means, the same DOM element may react to the same event type differently if its event handlers are changed.

In Section 6.3.1, we first explain different ways of registering event handlers to elements and later in Section 6.3.2, we explain how event identifiers are produced in our prototype.

6.3.1 Event Registration Methods

The evolution from simple HTML pages to RIAs has resulted in three different ways to register an event handler to a DOM element.

1. **As inline HTML:** The oldest way is to specify the event handler as an attribute of the HTML element. For example, the following shows a `div` element that reacts

to mouse click events.

```
<div id="id1" onclick="doSomething()">Text content</div>
```

The event handler is registered using the `onclick` attribute of the HTML element. The value of the attribute is the event handler, in this case a call to a JavaScript function named `doSomething`.

For an anchor element, `a`, it is also possible to use the `href` attribute to trigger JavaScript code when the anchor is clicked. This is achieved by specifying the event handler as the value of the `href` attribute as follows.

```
<a href="javascript:‘doSomething()’">Text content</a>
```

The `href` attribute is normally used to specify the URL that needs to be loaded when the anchor is clicked; however, browsers interpret a string that is prefixed by the keyword "javascript:" and that is used in a place where a URL is expected as JavaScript code.

2. **Assignment via JavaScript:** Later, browsers allowed registration of event handlers by assigning the event handler as a property of the DOM element through JavaScript. This method made dynamic registration of event handlers possible. The following example registers the function named `doSomething` as an `onclick` event handler to the element with id `id1`

```
<script type="text/javascript">
document.getElementById("id1").onclick = doSomething;
</script>
```

3. **Using DOM Event Specification:** The most recent and advanced way of event registration is through using the event registration model that is introduced in DOM Level 2 specification. Unlike the previous methods, which only allow a single event handler for each type of event, in this model, any number of event handlers can be added for each type of event using the `addEventListener` method via JavaScript. (In Internet Explorer, this specification was not implemented exactly until version 9; a slightly different interface was used.) When multiple event handlers are registered to an event type, all the registered event handlers are run one after the other when the event occurs. The following example registers two event handlers to the `onclick` event of the element with id `id1`.

```
<script type="text/javascript">
var element = document.getElementById("id1");
element.addEventListener = ("click", doSomething);
```

```
element.addEventListener = ("click", doAnotherThing);
</script>
```

In this example, when the element is clicked, both JavaScript functions, `doSomething` and `doAnotherThing`, will be executed. In this model, it is also possible to remove a previously registered event handler using the `removeEventListener` method.

6.3.2 Implementation

Our crawler supports the three types of event registrations methods mentioned. The AppScan component provides us the set of DOM elements that have registered event handlers. The event handlers that are registered using method 1 is easily detectable since they are part of the HTML. For DOM elements that have events registered using methods 2-3, AppScan uses its JavaScript Engine to keep track of which DOM elements have registered event handlers.

Once all the elements with event handlers are known, we need to generate an event identifier for each element and event type. Our identifiers have the following form:

`DOMELEMENTID~~EVENTHANDLERID~~EVENTTYPE`

where `~~` is a delimiter separating different information for the event. `DOMELEMENTID` is an identifier for the DOM element that the event is associated with, `EVENTHANDLERID` is an identifier for the event handlers (which is produced by hashing the definitions of the JavaScript functions registered for that event), `EVENTTYPE` is the type of the event.

In the current implementation, we have two different mechanisms to produce the `DOMELEMENTID` component of the event identifier. These two methods are explained next. Later, we explain the event types supported in our crawler.

Identifying DOM Elements (Generating `DOMELEMENTID`)

Method 1 - Element Identifier based on HTML: In this method, the DOM element identifier is produced based on the HTML representation of the DOM element. For example, consider the following HTML anchor element which has an event handler registered using the `href` attribute.

```
<a href="javascript:‘doSomething()’"><img src="something.jpg" /></a>
```

The produced event identifier for the `href` event of this element is

```
<a href="javascript:‘doSomething()’"><img src="something.jpg" /></a>~~~~HREF
```

In this event identifier, `DOMELEMENTID` is the HTML of the element itself. The `EVENTHANDLERID` is empty since the event handler `doSomething()` is part of the DOM element

identifier (The `EVENTHANDLERID` part contains a value only for the events registered using methods 2 and 3 from Section 6.3.1). The `EVENTTYPE` is `HREF` which shows that the element has an event registered to it using its `href` attribute.

In some cases, this element identification method might not be able to produce a unique identifier for an element. For example, Figure 6.2 shows the rendering of an HTML table in the browser on the left and the corresponding HTML body on the right. Let's assume that the following JavaScript is executed to register the `removeRow` function as an event handler on the third column of each row such that the row is removed from the table when the third column is clicked.

```
<script type="text/javascript">
removeRow = function(){
  this.parentNode.parentNode.removeChild(this.parentNode);
};
var rows = document.getElementsByTagName("tr");
for (var i=0; i < rows.length; i++){
  rows[i].getElementsByTagName("td")[2].addEventListener("click", removeRow);
}
</script>
```

The explained element identification method cannot differentiate between these elements since their HTML code are identical:

```
<td style="color:blue"> Remove </td>
```

Moreover, each element have the same event handler "removeRow". (The keyword "this" used inside the function is a reference to the element that owns the handler. With this reference the function is able to remove the row whose third column is clicked.) When an event cannot be identified uniquely as in this example, the crawler ignores the event.

A more powerful alternative for DOM element identification is explained next.

Method 2 - Element Identifier based on Element Characteristics and Neighborhood Influence: We have implemented an alternative method [60] to identify DOM elements. This new method is more powerful than the previous one at differentiating between the elements that have identical HTML code.

Figure 6.2: A page containing a table (left) and the body of the corresponding HTML document (right)

This alternative method consists of two phases. In the first phase, an "element characteristic identifier" is produced for each DOM element that has an event, using the following information about the element

- the tag name of the element (e.g., <td>, <a>)
- the name,value pairs for a chosen set of attributes. For example, a possible set may contain id, title, href, src, name, class, and the event attributes like onmouseover, onmouseenter, onclick etc.
- the text content of the element

If the element characteristics is enough to differentiate an element from the other elements in the DOM, then the element characteristic identifier is used as the identifier for the element. If there are multiple elements with the same element characteristic identifier, then the second phase of the algorithm tries to add some "neighborhood influence" to differentiate between such elements. In other words, for each element x in a set of elements that have identical characteristic identifiers, we check if some element that is

close to x in the DOM-tree can provide information that can allow us to differentiate x from the other elements in the set.

That is, given a set, $X = \{x_1, x_2, \ldots x_n\}$, of elements that have the same element characteristics identifiers, the second phase applies the following steps.

1. Pick an element from X, say x_1 (it does not matter which one is picked)
2. Taking x_1 as the starting point, start traversing the DOM in a Breadth-First manner and apply the following steps at each visited element (this traversal is not limited to the subtree of x_1: the ancestors of x_1 are also visited in addition to the descendants of x_1)
   (a) Let y be the currently visited element and let *relXPath* be the relative XPath from x_1 to y
   (b) For each element $x_i \in X$, identify the element z_i that is reached following the *relXPath* starting from x_i. If z_i does not exist for an x_i, continue the Breadth-First traversal with the next element in the DOM (go to step 2a). Otherwise, compute a new identifier for x_i by appending *relXPath* and the element characteristic identifier of z_i to the element characteristic identifier of x_i.
   (c) If the new identifier of each x_i is unique, then stop the Breadth-First traversal and use these new identifiers. Otherwise, continue traversal with the next element (go to step 2a).

As an example, consider the three elements with identical HTML code from Figure 6.2:

```
<td style="color:blue"> Remove </td>
```

The element characteristics ids for these three elements will be the same:

td,Remove

where comma is used as a delimiter between different components of a element characteristic id. For these elements, the following unique identifiers will be obtained after the second phase is applied.

td,Remove;./../*[1];td,id,1,1
td,Remove;./../*[1];td,id,2,2
td,Remove;./../*[1];td,id,3,3

Each id is obtained by combining the characteristic id of an element "td,Remove", the relative XPath "./../*[1]" (which points to the first sibling of the element in this case) and the characteristic id of the element that is pointed by the relative XPath in each case.

It is still possible that there can be a set of elements that cannot be uniquely identified even after the second phase. If that happens, we do not try any further and simply ignore these events.

Event Types

Our crawler currently supports the mouse events and the events which are defined on the `href` attribute of anchor tags using the "javascript:" prefix (which are also mouse events since they are triggered when the anchor is clicked).

The mouse events that are considered are mouseover, mouseenter, mousedown, mouseup, click, dblclick, mouseout and mouseleave. Instead of considering all these mouse events as individual events, we create "composite" events that execute these events in a specific sequence. The reason for doing this is to better simulate a user's behavior. For example, when a human user wants to click on an element in the browser, she has to first move the mouse over the element. This will trigger the mouseover and mouseenter handlers. Then, she will be able to click it. In addition, a mouse click event handler is only triggered after the mousedown and mouseup handlers are executed. It is also a typical user behavior to move the mouse away from the element once it is clicked and this will cause mouseout and mouseleave handlers to be triggered.

We have defined two composite mouse events: one is simulating a mouse interaction sequence without the double click event and the other simulating a double click. The first composite mouse event executes the following sequence: <mouseover, mouseenter, mousedown, mouseup, click, mouseout, mouseleave>. For an element to have this composite event, it is not necessary for the element to have all the mouse events in this sequence defined. For example, for an element that only has mouseover and mousedown defined, the composite event is still created; when the composite event is triggered, it will run first the mouseover handler and then the mousedown handler.

The second composite mouse event is a sequence that simulates a double click: <mouseover, mouseenter, mousedown, mouseup, click, mousedown, mouseup, click, dblclick, mouseout, mouseleave>. For an element to have this composite event, the element has to have a dblclick handler defined. Since all the other mouse handlers are already considered by the previous composite event, there is no point of having a second composite

event if the element does not have a dblclick handler.

6.4 DOM Equivalence

Our crawler uses AppScan's DOM Equivalence algorithm [12] with a slight modification. The original algorithm implemented in AppScan produces an identifier for a given DOM by only considering the underlying HTML structure without taking into account the events enabled in the DOM. Since this algorithm only considers the HTML structure, in the remainder we refer to the identifier produced by this algorithm as *HTML ID*. How this algorithm works is explained below.

As we discussed in Section 2.3, it is important to require that the DOMs in the same equivalence class have the same set of enabled events. For this reason, our DOM identifiers are the combination of the HTML ID produced by AppScan and an identifier generated for the set of enabled events in the DOM. The latter identifier is simply produced by first sorting the set of event identifiers enabled in the DOM into a list and then concatenating the individual event identifiers in the list.

6.4.1 Computing the HTML ID

The algorithm aims at identifying the pages with similar page structure by reducing the repeating patterns in a given HTML to reach a canonical representation of the HTML document such that the canonical representation will be the same for any other structurally equivalent document.

The motivation of the algorithm comes from the observation that HTML pages often contain sub-structures that are containers for similar items, such as lists and tables. Modifying the items in the container, such as adding/removing items or sorting the items in a different order, usually does not change the structure of the container. To enable the crawler to detect such containers, the algorithm looks for repeating patterns in a document. Then, it reduces these repeating patterns in order to recognize the same container even when the items in the container are modified.

For example, Figure 6.2 shows an HTML table that consists of rows, <tr> elements, and each row contains columns, <td> elements. When all the text and attributes are stripped from the HTML (leaving only the HTML tags), the subtree rooted at <tbody> looks like

```
<tbody>
          <tr><td></td><td><a></a></td><td></td></tr>
          <tr><td></td><td><a></a></td><td></td></tr>
          <tr><td></td><td><a></a></td><td></td></tr>
</tbody>
```

where each row <tr> follows the same pattern. The algorithm recognizes such patterns and reduces them. In this example, the subtree would be like the following after the reduction.

```
<tbody>
          <tr><td></td><td><a></a></td><td></td></tr>
</tbody>
```

By finding such repetitive patterns, the algorithm aims at dividing a page into its sub-structures. An element whose subtree follows a repeating pattern of child elements is considered as a sub-structure since the repetition is seen as an indication that the element is a container for similar items. By reducing the repetitions, the crawler can recognize the same sub-structure even if the number of items contained in it changes. In addition, once all the reductions are done in a subtree, the algorithm sorts the remaining tags so that the algorithm is not affected by reordering of the elements inside a sub-structure. As a result, the produced identifier for a page will not change when a sub-structure is modified by adding or removing items in it or when its items are presented in different orders.

The algorithm allows the user to configure which HTML tags and attributes to consider, as well as whether to include the text content for the computation of the HTML ID. When provided an HTML page, the algorithm produces the identifier by applying the following steps:

1. The HTML is stripped out of anything that is not included in the user configuration.
2. Algorithm identifies a parent node whose children are all leaf nodes in the tree.
3. Algorithm traverses the leaf nodes and at each leaf node, it checks if the sequence of already traversed leaves and the current leaf node forms a pattern. A pattern is detected if the sequence contains consecutive repeating elements. For example, the sequence <A><B><C><A><B><C> contains the consecutive repeating pattern

<A><B><C>, whereas <A><B><C><D><A><B> has no consecutive repeating pattern. Although <A><B> is repeated, the repetition is not consecutive.

4. When such a repeating pattern is detected, all the repetitions are eliminated.

5. When the last leaf node of the parent is processed, the reduced sequence is sorted and the parent node is turned into a leaf node containing the reduced sequence. For example, when the last leaf in the leaf node sequence <B><A><C><B><A><C> of the parent <Parent> is processed, the result would be a new leaf node <Parent><A><B><C></Parent>
(i.e. a leaf node <Parent> with text "<A><B><C>").

6. Steps 2-5 are repeated until the stripped HTML is reduced to a single node. At this point, the resulting node uniquely identifies the equivalence class of the HTML page. By hashing the content of the node, the HTML ID is produced.

The configuration we used for the experimental study presented in this thesis includes the text content, all HTML tags and none of the attributes.

6.5 Conclusion

The crawler used in this research has been built as a prototype of IBM Security AppScan. To build the prototype, we have used some of the existing components of the AppScan. These are the DOM equivalence algorithm, the embedded browser, and the ability to detect elements which have registered events. On top of these functionalities, we have implemented the crawling strategies (except for the Greedy and the Menu, all strategies are implemented by me). In addition, the algorithms to produce event identifiers are implemented by me as part of this research.

Chapter 7

Experimental Results

7.1 Introduction

In this chapter, we present our experimental results comparing the performances of the crawling strategies for RIAs. The experiments were conducted using five real AJAX-based applications and three test applications.

For each application and for each strategy, we present two sets of measurements:

1. the cost (time) to discover the states of the application,
2. the cost (time) to complete the crawl.

We are primarily interested in the first set of measurements. According to our definition of strategy efficiency (explained in Section 1.3.3), the first set of measurements show how efficient the strategy is for an application. However, (as we discussed in Section 1.3.5) we have to crawl each application completely to obtain the first set of measurements since the crawler cannot know for sure that all the states are discovered until all transitions are taken at least once.

In addition to comparing the performances of strategies with each other, we also present the optimal cost to discover all the states in each application. This optimal cost is obtained after the model of the application is known.

The remainder of this chapter is organized as follows. In Section 7.2, we define our cost metric used to measure efficiency. In Section 7.3, we list the strategies with which we compare the Hypercube and the Probability strategies and explain how the optimal cost is obtained. In Section 7.4, our subject applications are introduced. In Section 7.5, we explain the details of how the experiments were conducted and how the correctness

of the produced models were verified. The costs of discovering states are presented in Section 7.6 and the costs for completing the crawl are presented in Section 7.7. In Section 7.8, we present the time measurements. Finally, Section 7.9 concludes the chapter.

7.2 Measuring Efficiency

We define the efficiency of a strategy in terms of the time the strategy needs to discover the states of the application. Instead of time measurements, we usually assess the efficiency of a strategy by the number of events executed and the resets used by the strategy during the crawl. This is because, the time measurements also depend on factors that are external to the crawling strategy, such as the hardware used to run the experiments and the communication delays, which can be different in different runs. In addition, the event executions and resets normally dominate the crawling time and they only depend on the decisions of the strategy. Nevertheless, we also provide time measurements in Section 7.8.

7.2.1 Cost Calculation

We combine the number of resets and the event executions used by a strategy to define a cost unit as follows.

- We measure for each application:
  - $t(e)_{avg}$: the average event execution time. This is obtained by randomly selecting a set of events in the application, measuring the execution time of each event in the set by simply executing the event once, and taking the average of the measured times.
  - $t(r)_{avg}$: the average time to perform a reset. This is obtained by loading the URL of the application multiple times: the time for each reload is measured and the average is taken.
- For simplicity, we consider each event execution to take $t(e)_{avg}$ time and use this as a cost unit.
- We calculate "the cost of reset": $c_r = t(r)_{avg}/t(e)_{avg}$.
- Finally, the cost of 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 a strategy to find all the states, respectively[1].

7.3 Strategies Used for Comparison and the Optimal Cost

We compare the model-based crawling strategies defined in this thesis (i.e., Hypercube and Probability) with the following strategies.

- **The Breadth-First and Depth-First Strategies:** These are the standard crawling strategies. The Breadth-First strategy explores the states in the order they are discovered (i.e., the state discovered earlier is explored first). The Depth-First strategy explores the states in the reverse order of their discovery (i.e., the most recently discovered state is given priority). To have a fair comparison with other strategies, our implementations of the Breadth-First and the Depth-First are optimized, so that the shortest known transfer sequence is used to reach the state where an event will be explored. The "default" versions of the Breadth-First and the Depth-First (simply resetting to reach a state) fare much worse than the results presented here.

- **The Greedy Strategy:** The Greedy strategy [62] is a simple strategy that prefers to explore an event from the current state, if the current state has an unexplored event. Otherwise, it explores an event from a state that is closest to the current state.

- **The Menu Strategy:** The Menu strategy [26] is another model-based strategy (see Section 5.2 for an overview).

- **The Optimal Cost to Discover All States:** We also present the optimal cost required for discovering all the states of an application. The optimal cost can only

[1]We measure the cost of reset before crawling an application and provide it 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 transferring from the current state to another known state. Although our measurements on the test applications show that the cost of reset is greater than 1, it does not mean it cannot be less than or equal to 1.

be calculated after the model of the application is obtained and can only be used as a benchmark. To calculate the optimal cost, we solve the Asymmetric Traveling Salesman Problem (ATSP) which is the problem of finding a path that visits all the nodes of a known graph with the minimum cost. We use an exact ATSP solver [22] to get an optimal path.

7.4 Subject Applications

In this experimental study, we used five real AJAX applications and three test applications. There are several difficulties that prevented us from using a larger number of real applications in this study: Our crawler, like the other available tools, is an experimental crawler. Our crawler implements its own embedded browser, and this has very important advantages that enable our crawler to overcome some of the limitations of the crawlers that work by driving external browsers[2]. However, our crawler's browser implementation uses some stub methods which are sometimes insufficient to realize the JavaScript functionality required by the RIA. In such cases, an effort is required to make necessary adjustments to crawl the application. This effort is not related to the crawling strategies explained in this thesis, but to execute the client-side code of the application correctly. We often need to improve the crawler to handle the JavaScript execution correctly, or if possible, we modify the JavaScript code of the application by replacing the JavaScript functionalities that are not supported by our crawler with alternative ones that work with our crawler and still allows the application function correctly.

Another difficulty with using real applications is that we need a local version of each application for crawling. We do not want to stress a publicly available application with the large amount of requests generated during the crawl since this may degrade

[2]An important task for a RIA crawler is to detect the DOM elements that have registered event handlers. When event handlers are registered dynamically using JavaScript (with addEventListener method explained in Section 6.3.1), unless the crawler itself implements a browser (and thus maintains the event handlers itself), it is currently not possible for the crawler to automatically detect if an element has such an event handler or not. This is because, currently this information is not accessible through the DOM interface. Since our crawler implements its own browser, it can detect such events automatically at run-time. But, for the tools which rely on external browsers, this is not possible. For example, Crawljax[55] is based on Selenium WebDriver (http://docs.seleniumhq.org/projects/webdriver/) which is an API to simulate user actions on a real browser. Since Crawljax depends on an external browser, it cannot detect such events automatically; the user needs to explicitly configure Crawljax by specifying which elements to interact with in an application.

the availability of the application for its real users and be considered a denial-of-service attack against the server. Also, the applications that are available on the public domains may change over time. This may prevent the reproducibility of the experiments in the future. For these reasons, we use applications that are either open-source or that we can replicate on a local server.

We describe the real applications in Section 7.4.1, followed by the test applications in Section 7.4.2. Table 7.1 shows the number of states, the number of transitions, and the measured cost of reset for each application.

	Name	Number of States	Number of Transitions	Cost of Reset
Real Applications	Bebop	1,800	145,811	2
	Elfinder	1,360	43,816	10
	FileTree	214	8,428	2
	Periodic Table	240	29,034	8
	Clipmarks	129	10,580	18
Test Applications	TestRIA	39	305	2
	Altoro Mutual	45	1,210	2
	Hypercube10D	1,024	5,120	3

Table 7.1: Subject Applications

7.4.1 Real Applications

Bebop

Bebob[3] is a real application which allows browsing a list of publication references. As shown in Figure 7.1, the top portion of the application contains a set of events for filtering the displayed references according to different categories (by year, by document type, by author etc.). The references are displayed at the bottom the page. For each reference, the names of the authors and the publication year also act as filters (i.e., when they are clicked, the reached page shows the publications of the author or the publications in that year). When the title of a reference (or the arrow icon under a reference) is clicked, more details about the reference are shown. This also enables another event which shows the Bibtex entry for the reference. For experiments, we used a version of the application where the total number of references that can be listed is 5. The application has 1,800

[3]http://people.alari.ch/derino/Software/Bebop/index.php
(Local version: http://ssrg.eecs.uottawa.ca/bebop/bebop/)

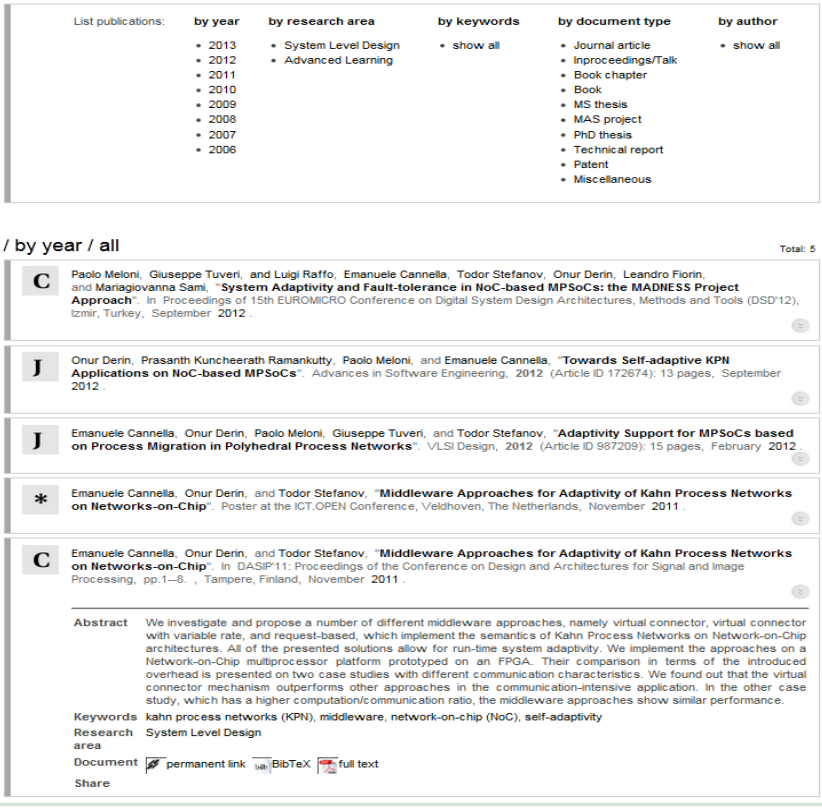

Figure 7.1: A state in Bebop where all publications are listed and the details of one of the publications has been expanded.

states and 145,811 transitions. The measured cost of reset is 2.

Elfinder

Elfinder[4] is a real application in the form of an AJAX-based file manager. The application allows browsing folders and files in a file system. For this experimental study, we used a simplified version of this complex application. The version we crawled contains (under the root folder) a single folder with three image files in it. Even for such a small instance, there are 1,360 states and 43,816 transitions. To be able to crawl this application, we have disabled the functionalities that change the server-side of the application, such as modifying existing files and folders (editing, renaming etc.) and creating new items etc. As it can be seen in Figure 7.2, the navigation is achieved by using the file tree on the left or double-clicking on the folder icons on the right. In addition, up and home buttons on

[4]http://elfinder.org/ (Local version: http://ssrg.eecs.uottawa.ca/emre/elfinder/)

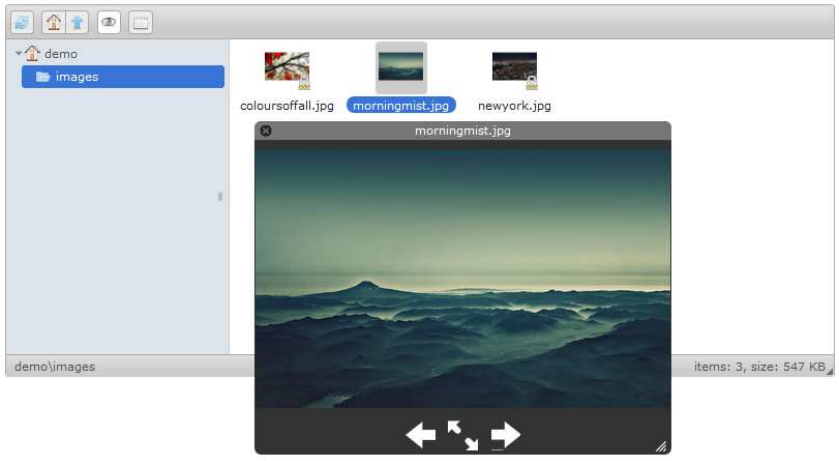

Figure 7.2: A state in Elfinder where the preview of the selected item is shown in the preview window.

the toolbar can be used for going to the parent folder and going back to the root folder, respectively. The view button on the toolbar switches how the items are displayed: as large icons, or as small icons in a list. Also on the toolbar, there is a preview button that shows the preview of a selected file or folder. The preview window can be switched between full screen and windowed mode. Using the previous and next buttons on the preview window, the previews for the next and the previous items in the current folder can be displayed. The measured cost of reset is 10.

FileTree

FileTree[5] is a real application which allows navigating a folder structure. As shown in Figure 7.3, clicking on a folder expands the folder (shows the folders and the files contained in the clicked folder). For this study, we used a version that uses the directory structure of the Python source code. The application has 214 states and 8,428 transitions. The measured cost of reset is 2.

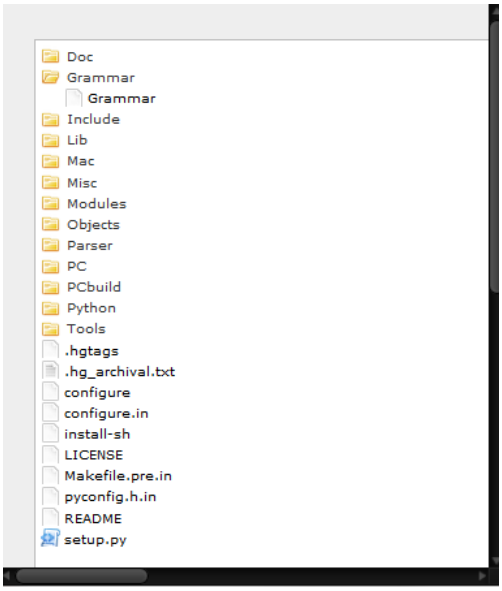

Figure 7.3: A state in FileTree

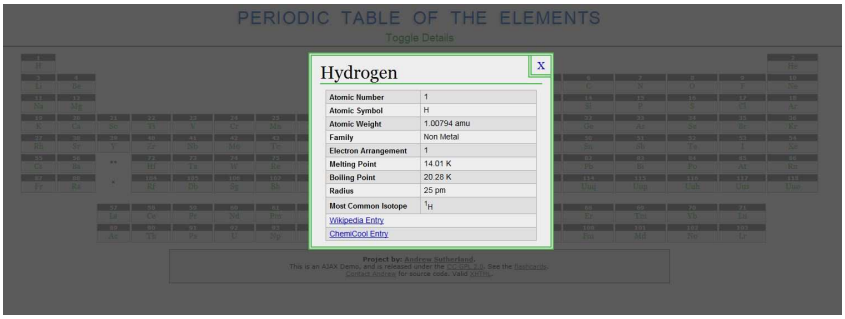

Figure 7.4: The state reached after clicking Hydrogen in Periodic Table

Periodic Table

Periodic Table[6] is a real application which is an AJAX-based periodic table. It contains the 118 chemical elements in an HTML table. As shown in Figure 7.4, clicking on an element in the table displays some information about the chemical element in a window. At any time, the window contains only the information of the last clicked element. Except for the initial state, each of these states is reachable from any other, thus forming a complete graph. Also, there is an event at the top of each page (Toggle Details)

[5]http://www.abeautifulsite.net/blog/2008/03/jquery-file-tree/
(Local version:http://ssrg.eecs.uottawa.ca/filetree_python/)

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

which switches the style of the current page between two alternative styles. This creates another complete graph such that two complete graphs are connected to each other with the instances of this toggle event. The application has 240 states and 29,034 transitions. The measured cost of reset is 8.

Clipmarks

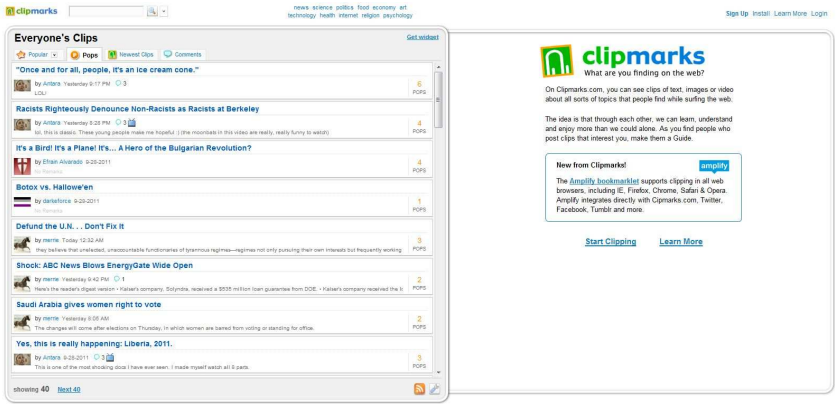

Figure 7.5: The Initial State of Clipmarks

Clipmarks[7] is an AJAX-based real application which allows its users to share parts of any webpage (images, text, videos etc.) with other users. For this experimental study, we used a partially replicated copy of the application. The initial page (shown in Figure 7.5) contains on the left hand side a list of clips which have recently been "popped" (voted as worth seeing) by other users. For each clip in the list, the title of the clip, the user sharing the clip, and the number of pops clip received are displayed. By clicking on the number of pops for a clip, the list of users who voted for the clip displayed in a dialog window. Clicking on the title of a clip item in the list loads the content of the clip on the right hand side of the page. A user may choose to share the displayed clip on other social networking sites, start following the user who posted the clip, or pop the clip. Each of these actions opens/changes the content of the dialog window. For this experimental study, we limited the number of clips in the list to three. The application has 129 states and 10,580 transitions. The measured cost of reset is 18.

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

7.4.2 Test Applications

TestRIA

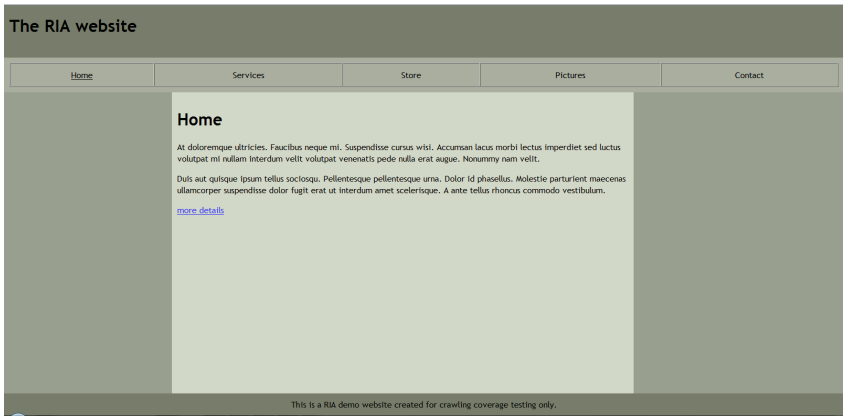

Figure 7.6: The Initial State of TestRIA

TestRIA[8] is an AJAX-based test application created by our research group. The application (shown in Figure 7.6) is in the form of a typical company website (or a personal homepage). Each state of the application contains 5 menu items at the top: Home, Services, Store, Pictures, Contact. Each of the menu items leads to a different section of the website. Home leads to the initial state and Services leads to a page where the services offered are listed as a side menu. Store leads to a page where the user can navigate the products the company offers. Pictures leads to a page where two photo albums can be viewed. In addition to the menu events, the application also contains the common previous/next style navigation in its Store and Pictures sections. TestRIA has 39 states and 305 transitions. The measured cost of reset is 2.

Altoro Mutual

Altoro Mutual[9] is a test application for a fictional bank (shown in Figure 7.7). Originally, this is a traditional web application developed by the AppScan team, but we created an AJAX version of the website where each hyperlink is replaced with a JavaScript event that retrieves the content using AJAX. The application has 45 states and 1,210 transitions. The measured cost of reset is 2.

[8] http://ssrg.eecs.uottawa.ca/TestRIA/index.htm
[9] http://altoromutual.com

Figure 7.7: The Initial State of Altoro Mutual

Hypercube10D

Hypercube10D is an AJAX test application which has the structure of a 10 dimensional hypercube (shown in Figure 7.8). This application represents the best case for the Hypercube strategy. The application has 1,024 states and 5,120 transitions. The measured cost of reset is 3.

Figure 7.8: The Initial State of Hypercube10D

7.5 Experimental Setup

We run each strategy 25 times on each application. In each run, the events of each state are randomly shuffled before they are passed to the strategy. The presented results are

the average of these runs. The aim here is to eliminate the influence of exploring the events of a state in a certain order. This is because, a strategy which does not have an exploration priority for the events on a state (this is always the case for the Breadth-First, the Depth-First and the Greedy) explores the events of the state in the order the events are provided to the strategy, which may improve or worsen the time required to infer the model. We would like to minimize the effect of this order.

To reduce the time it takes to run an experiment, we simulated the strategies on the applications' models. That is, we first crawled each application to obtain the model of the application and saved this model in the hard drive. During a simulation, we feed this model to a strategy. From the point of view of the crawling strategy, there is no difference between a simulation and an actual crawl (i.e., the same implementation of each crawling strategy can be used for a simulation and an actual crawl). We used this approach since it takes very long time to crawl some of the subject applications with the standard strategies, especially Depth-First. For example, it takes 4 days to crawl Bebop with the Depth-First strategy. Simulations save us time since there are not any JavaScript execution and communication delays when running a simulation. Simulations are enough to obtain the information required to compute the cost of each crawl: the number of events executed and the number of resets. However, we also provide the time measurements which are obtained by actually crawling each application with each strategy once.

In Chapter 6, we explain the DOM equivalence and the event identification mechanisms used by our crawler. We used element identifiers based on HTML code (i.e., the first method explained in Section 6.3.2) to produce event identifiers for all subject applications, except for Elfinder and Bebop. For Elfinder and Bebop, element characteristics and neighborhood influence method (i.e, the second method explained in Section 6.3.2) was used since these applications contain events which cannot be differentiated by the first event identification method.

To verify that the model extracted by our crawler is a correct model of the application behavior, we manually checked the states and transitions in the model against the observed behavior of the application in an actual browser. To ease this verification process, we have developed a model visualization tool which can represent the extracted model as a directed graph (shown in Figure 7.9). The tool also allows us to replay in an actual browser the transitions the crawler traversed to reach a state.

For the Probability strategy, the initial probability is taken as 0.75 by setting $p_s = 0.75$ and $p_n = 1$. This value has been chosen, because it gives slightly better results than the

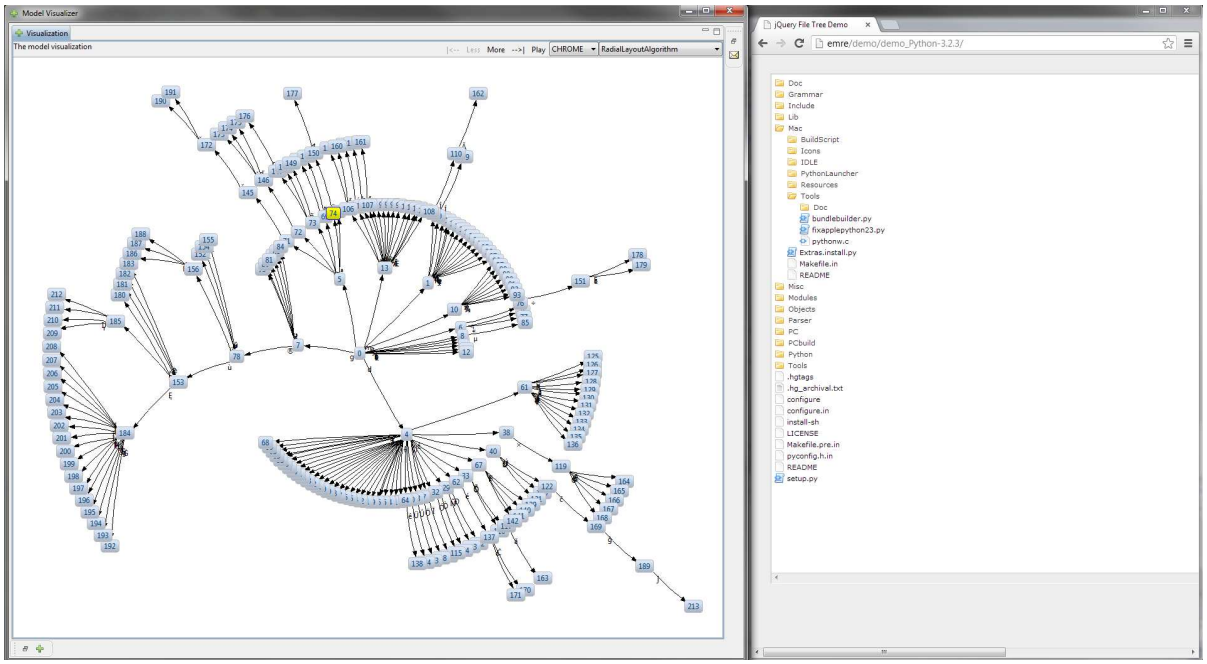

Figure 7.9: On the left, the visualization tool shows the extracted model of FileTree partially (the tool is configured not to show all the transitions for a clearer picture). When a node is clicked on the graph, the state corresponding to the clicked node is opened in a real browser as shown on the right. This state is reached automatically by replaying the events taken by the crawler in the browser.

other values we have experimented with (the results when other initial probabilities are used are given in Section A.4 of Appendix A).

7.6 State Discovery Results

In this section, we compare the efficiencies of the strategies using our cost metric defined in Section 7.2.1. According to the our definition of the efficiency, a strategy is more efficient if it discovers the states earlier. For this reason, the costs we present in this section are the costs up to the point when all the states of an application are discovered. Discovering all the states does not mean the termination of the crawl since the crawl continues until all the events are explored. The costs at the time of termination are presented in Section 7.7.

For each application and strategy, we present a plot that shows the costs required by the strategy to discover the states of the application. The x-axis of the plot is the number

of states discovered and the y-axis is the cost. A point (x, y) in the plot of a strategy means that when the number of states discovered by the strategy reached x, the total cost of the crawling since the beginning was y. Since the states may not be discovered in the same order by the strategies, the x-th states discovered by different strategies are not necessarily the same. These plots help us to see a strategy's progress during the crawl and determine how quickly the strategy discovers the states. For a better presentation, we use logarithmic scale. The cost required by a strategy to discover all the states is shown next to the last point of the corresponding plot.

The results show that model-based crawling strategies and the Greedy strategy are significantly better than the Breadth-First and the Depth-First. In addition, the Menu and the Probability are comparable to each other and are the most efficient of all in most cases. The Hypercube strategy behaves very similar to the Greedy strategy when the application does not follow the hypercube model at all. This is not surprising, given the fact that the Hypercube strategy always tries to explore an event from a state that is closer to the current state when the application does not follow the Hypercube meta-model. That means, the Hypercube strategy falls back to a Greedy strategy when the application does not follow the meta-model. However, there is a slight difference between the Greedy strategy and the Hypercube. Between two states that are at the same distance from the current state, the Hypercube strategy prefers to explore an event from the state that has larger number of events (in the same case, the Greedy strategy has no preference). This is because, according to the Hypercube assumptions, the state with more enabled events have a larger anticipated model, which means more chances of discovering a new state.

7.6.1 Bebop

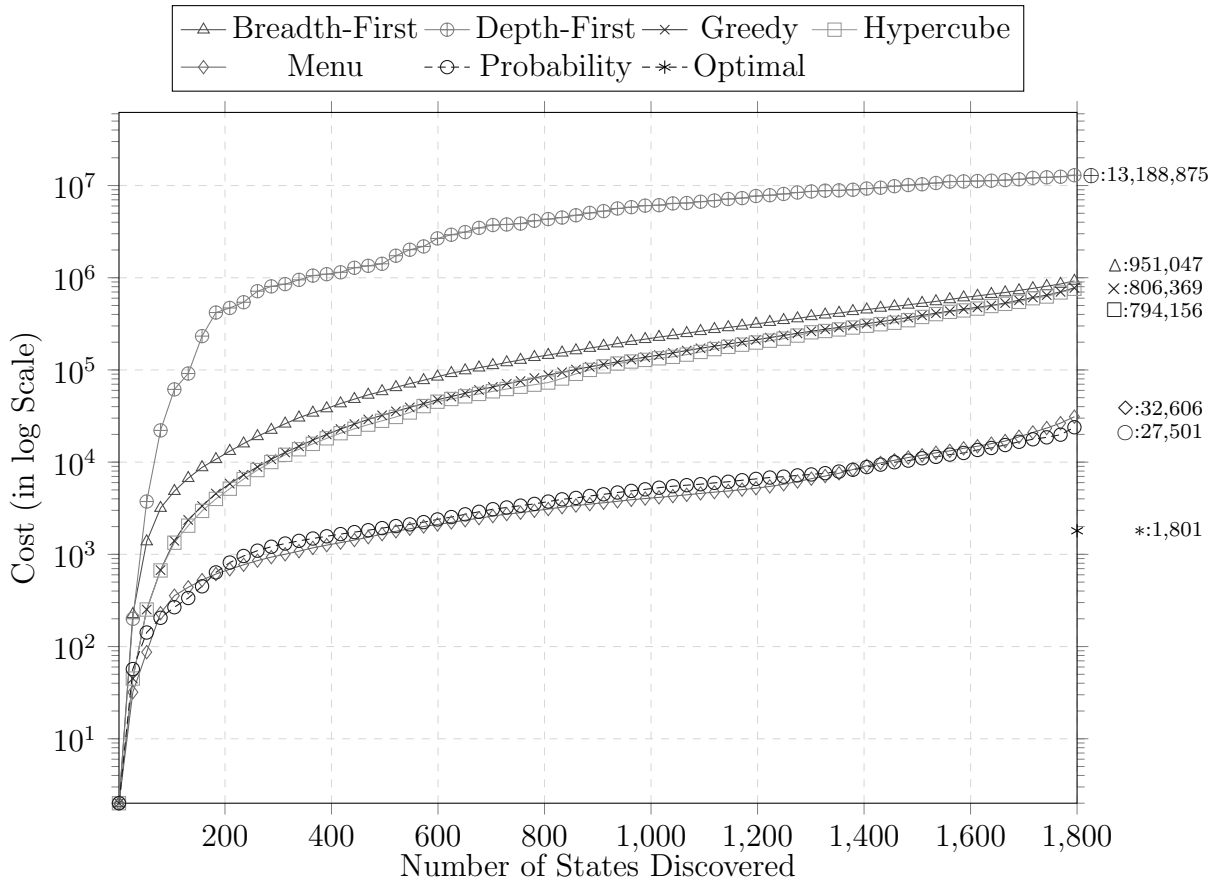

Figure 7.10: State Discovery Costs for Bebop (in log scale)

The state discovery plots for Bebop are shown in Figure 7.10. We see that the Probability and the Menu are significantly better than the other strategies. The Hypercube and the Greedy show very similar performances. The Breadth-First strategy follows these. The Depth-First strategy is significantly worse than all the other strategies.

7.6.2 ElFinder

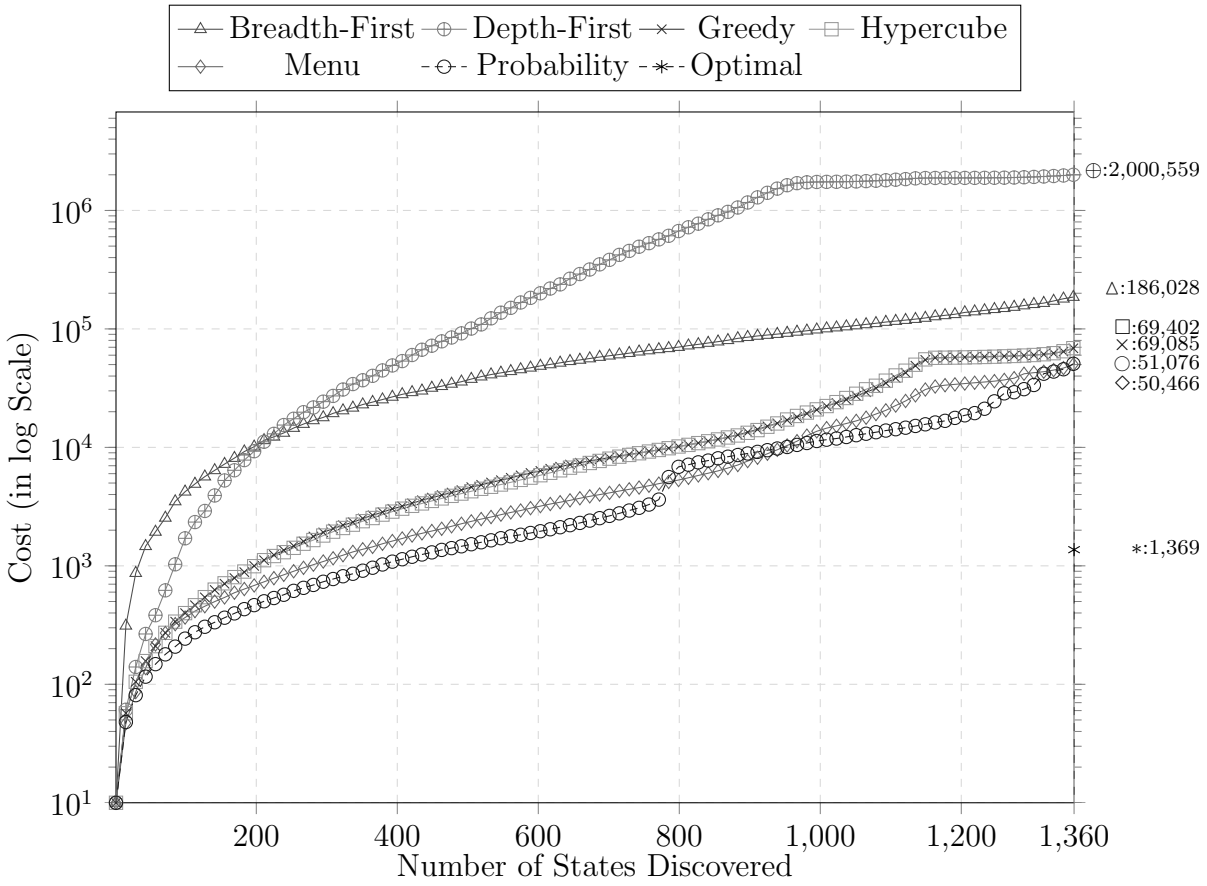

Figure 7.11: State Discovery Costs for ElFinder (in log scale)

The state discovery plots for Elfinder are shown in Figure 7.11. The Probability and the Menu perform similarly and better than the others. The Hypercube and the Greedy show similar performances to each other. These are followed by the Breadth-First strategy. The Depth-First strategy is significantly worse than the others.

7.6.3 FileTree

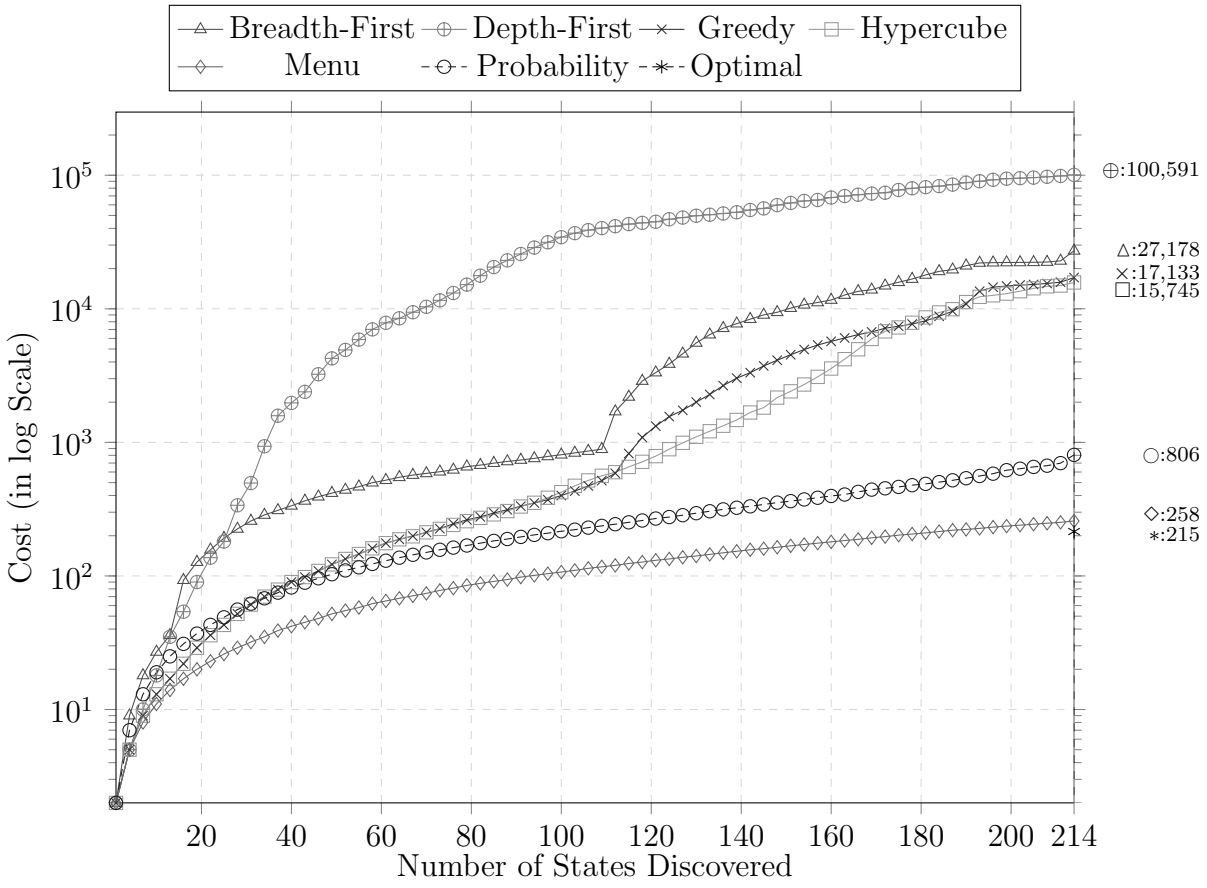

Figure 7.12: State Discovery Costs for FileTree (in log scale)

The state discovery plots for FileTree are shown in Figure 7.12. This application is special in the sense that a new state is discovered when an event is explored for the first time (a folder is expanded), but none of the subsequent explorations of the event produce a new state. All the states in this application can be discovered by simply exploring each event once. This is what is done by the Menu strategy initially: the Menu strategy gives the highest priority to events that has not been explored at all. For this reason, the Menu strategy performs close to optimal. The Probability strategy follows Menu and it is significantly better than the rest. The difference between the Probability strategy and the Menu is explained by the fact that in this application, the Probability strategy prefers an event that is explored once over an event that is not explored yet (since an event that is explored once has a higher probability). The Hypercube and the Greedy

show similar performances and they are followed by the Breadth-First. The Depth-First strategy is significantly worse than the other strategies.

7.6.4 Periodic Table

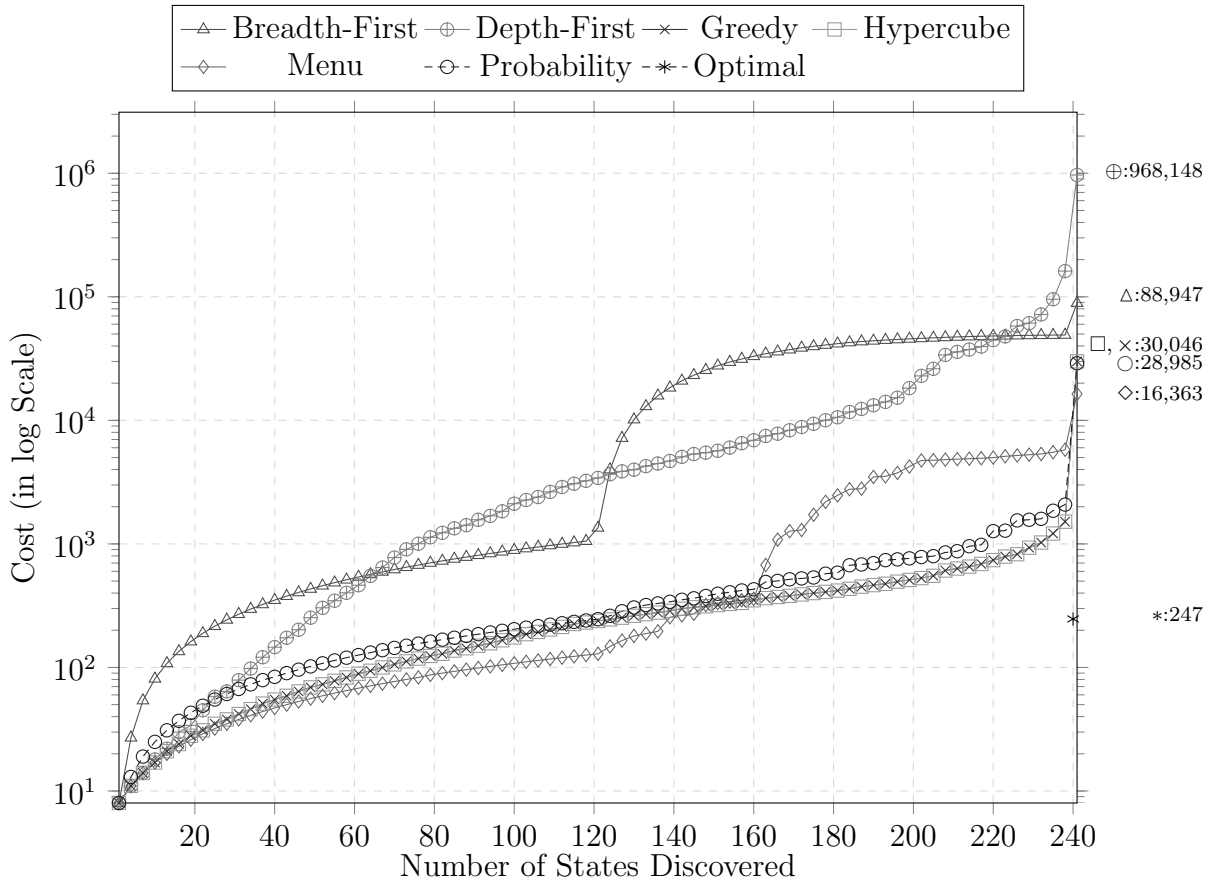

Figure 7.13: State Discovery Costs for Periodic Table (in log scale)

The state discovery plots for Periodic Table are shown in Figure 7.13. It can be seen that Hypercube, Probability, Menu and Greedy are more efficient than the Breadth-First and the Depth-First. In this application, the first exploration of each event corresponding to a chemical element results in a new state and all these events are present in any state. Thus, a strategy that clicks on each element once easily discovers the half of the states. For this reason, the Menu strategy, which explores each event once first, discovers half of the states much faster than the others. Note that, Breadth-First also first explores all

the events in the initial state and by doing so discovers half of the states. However, the Breadth-First needs a reset before each event exploration to go back to initial state and that increases the cost.

The Hypercube and Greedy strategy have very similar performances. They are faster than the Menu strategy once half of the states are discovered.

At the beginning, the Probability strategy is a bit slower compared to Greedy, Hypercube and Menu, but catches up once half of the states are discovered. The reason for the difference at the beginning is that for this application, the Probability strategy prefers events that are explored once over the events that are not explored yet since the first exploration of each event leads to a new state. Thus, the Probability strategy explores an event at least twice at the beginning, before trying an event that is not explored at all (whereas the Greedy and Hypercube strategies pick any event). But, in the majority of the cases the second exploration of an event do not lead to new states.

A peculiarity of this application is that each strategy discovers the last state much more later than the rest of the states (i.e., the last point in each plot is significantly higher than the preceding data point in the plot). This is because, the application has a state that can only be reached through the initial state and the initial state is not reachable from the other states unless a reset is used. Except for Breadth-First, all the strategies go back to initial state towards the end of the crawl, once all the other states are explored. Hence, the mentioned state is discovered much later than others. Since Breadth-First explores all the events in the initial state first, it is not affected. However, in the case of Breadth-First, there is still a noticeable gap between the last state and the preceding state. This is explained by the fact that all states in this application is at most two event executions away from the initial state, except for one that requires three event executions. Since this state is at one level deeper than the rest of the states, it is the last state discovered by the Breadth-First.

7.6.5 Clipmarks

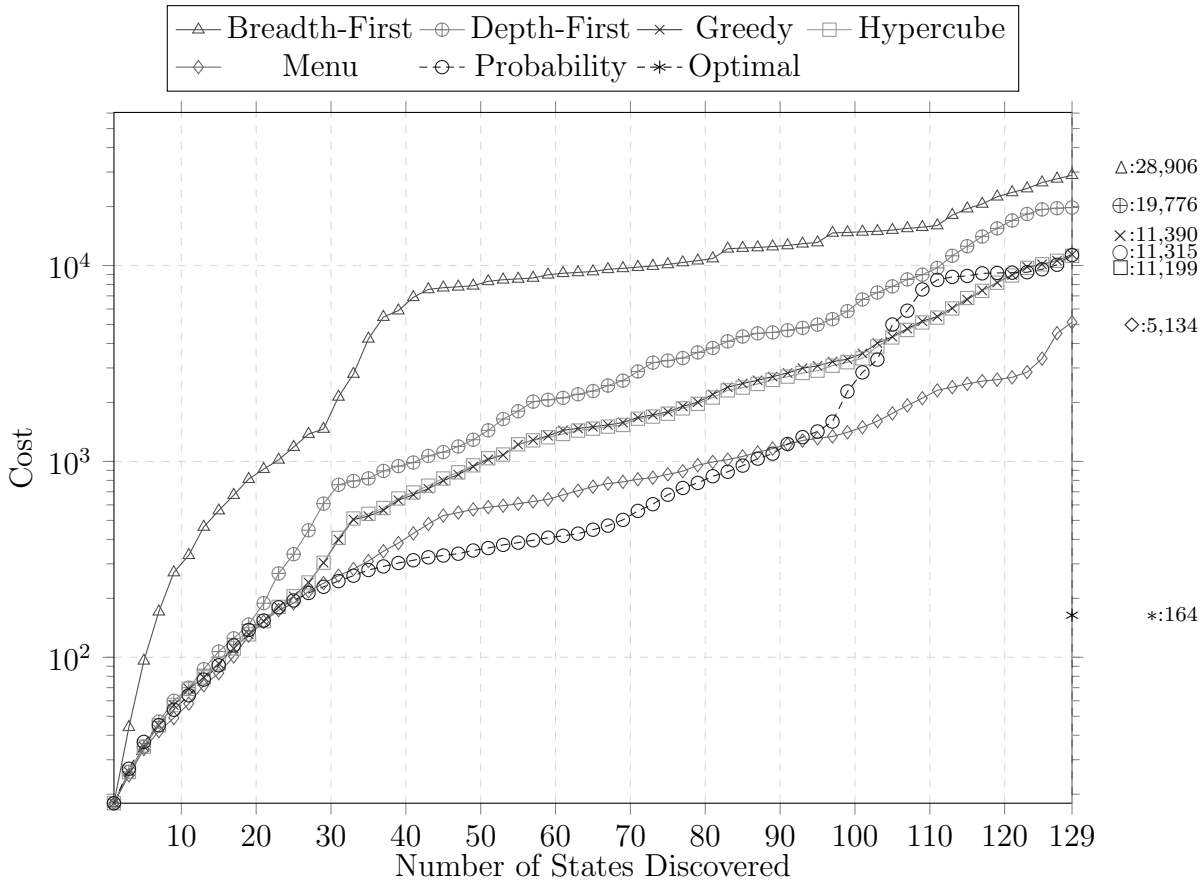

Figure 7.14: State Discovery Costs for Clipmarks (in log scale)

The state discovery plots for Clipmarks are shown in Figure 7.14. The Menu and the Probability strategies show comparable performance while Probability is slightly better for the most part, it gets worse towards the end. The reason is that this application has quite a large number of self-loop type events and the Menu strategy is able to classify them correctly and postpone their exploration towards the end of the crawl. The Hypercube and Greedy strategies have similar performances and they follow the Menu and the Probability. The Depth-First and the Breadth-First show the worst performances.

7.6.6 TestRIA

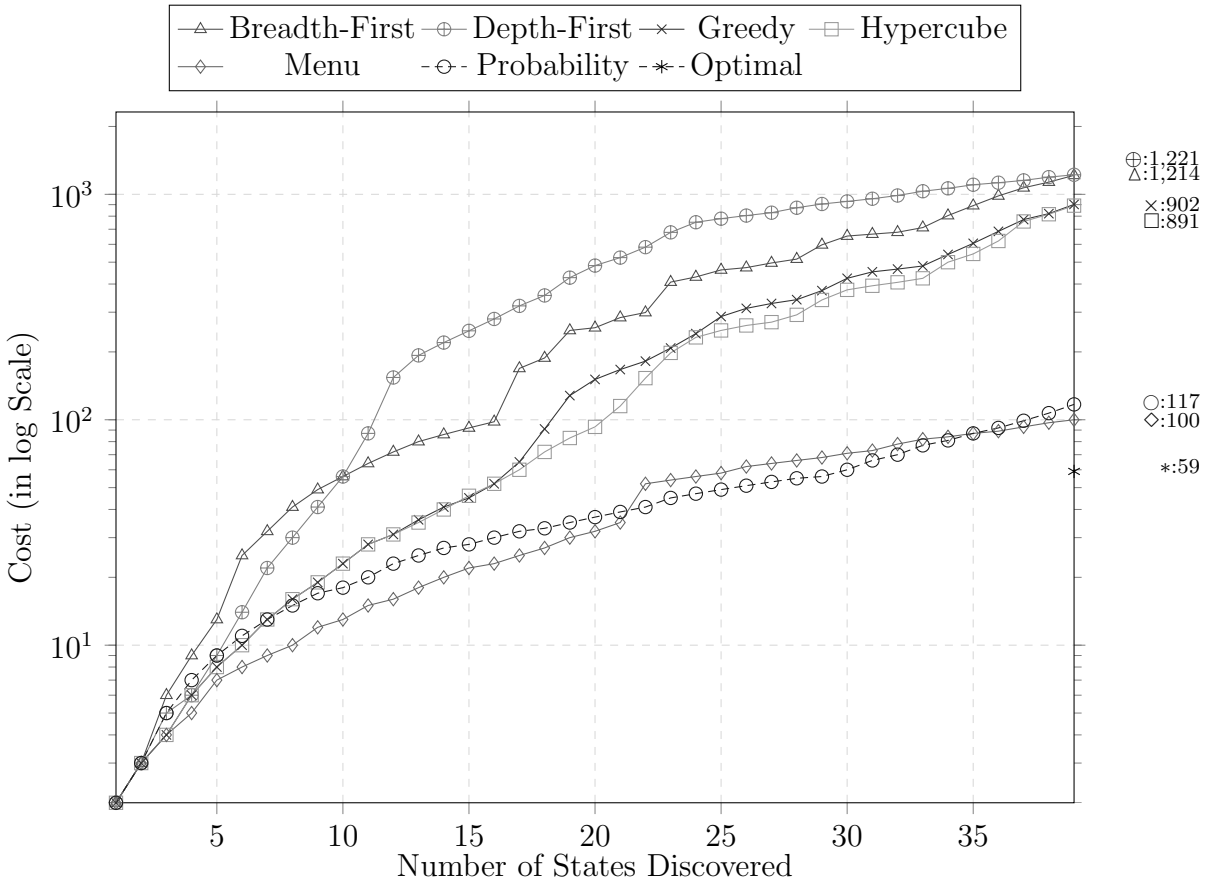

Figure 7.15: State Discovery Costs for TestRIA (in log scale)

The state discovery plots for TestRIA are shown in Figure 7.15. The Hypercube and the Greedy show similar performances and they are significantly better than the Breadth-First and Depth-First. The Probability and the Menu show the best performance and one is not significantly better than the other.

7.6.7 Altoro Mutual

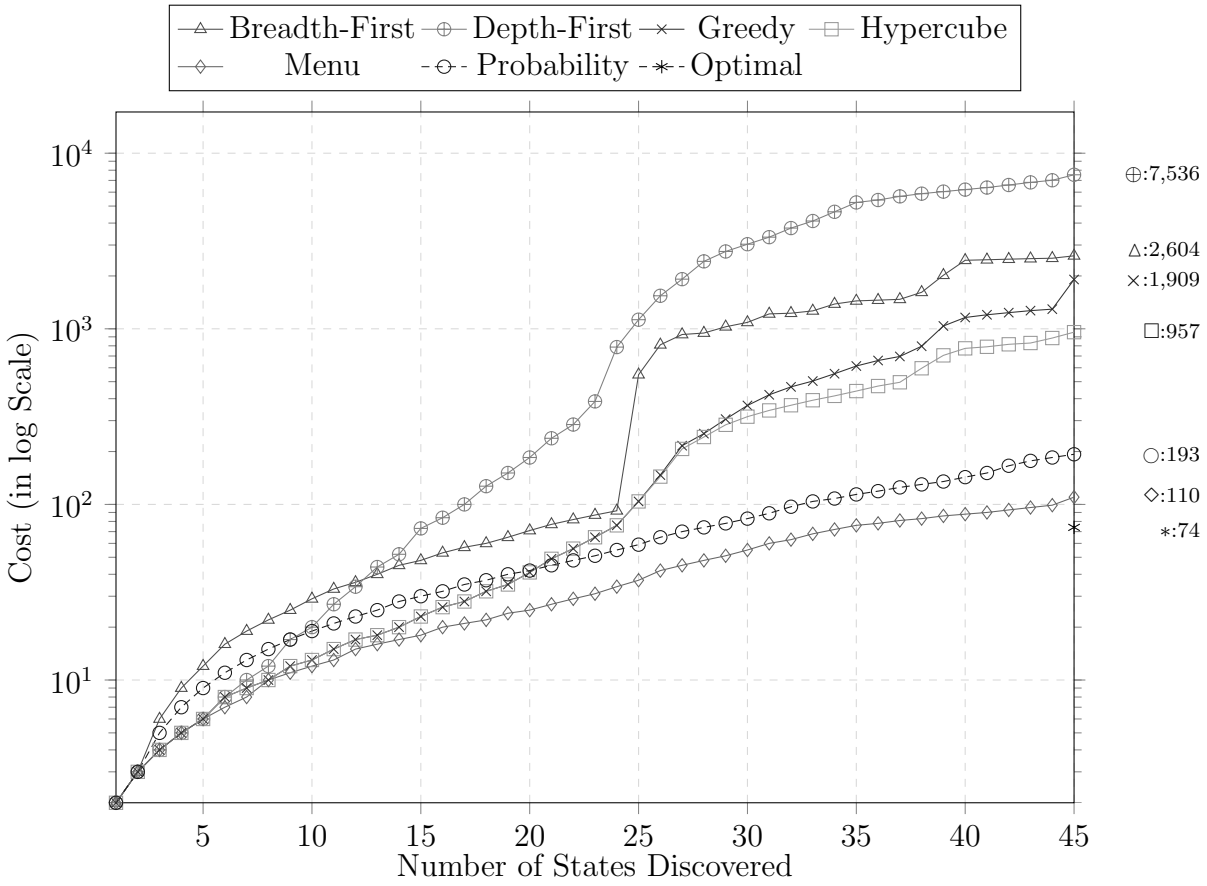

Figure 7.16: State Discovery Costs for Altoro Mutual (in log scale)

The state discovery plots for the Altoro Mutual are shown in Figure 7.16. This application is the best case for the Menu strategy as it consists of Menu type events only. As expected, Menu shows the best performance. The Probability strategy shows a comparable performance to the Menu strategy. The Hypercube strategy follows the Probability and it is better than Greedy strategy (preferring states with more events seems to be advantageous in this application). The Depth-First and the Breadth-First are again significantly worse than other strategies.

7.6.8 Hypercube10D

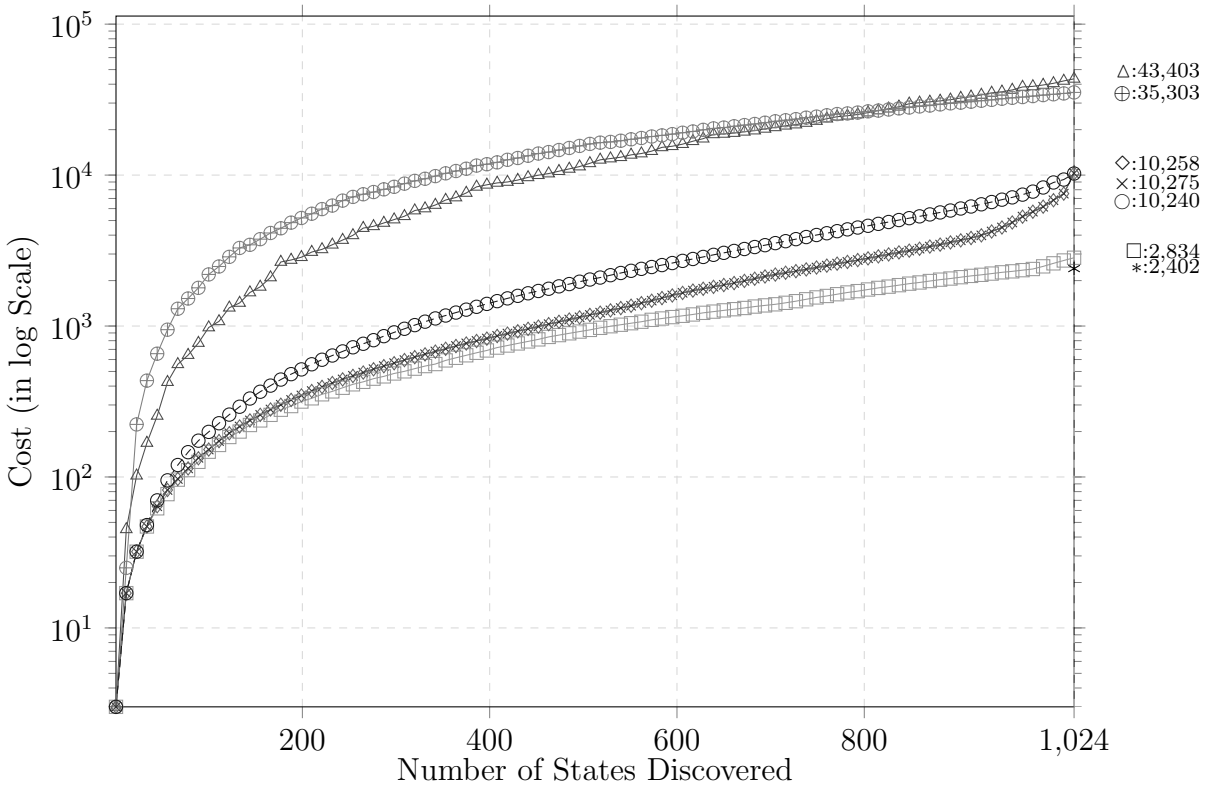

Figure 7.17: State Discovery Costs for Hypercube10D (in log scale)

The state discovery plots for Hypercube10D are shown in Figure 7.17. This application is the best case for the Hypercube strategy. For this application, there is a slight difference between the optimal cost and the Hypercube strategy's cost. As explained in Section 4.4.5, the Hypercube strategy deliberately executes more events during state exploration to keep the total number of resets for the complete crawl minimal. (This is the number of resets required not for discovering all the states, but for exploring all the transitions. This number is presented in 7.4 in the next section.) The number of resets Hypercube strategy uses to discover all the states (the number given in Table 7.2) is not affected (i.e., it is optimal).

Unsurprisingly, the Hypercube shows the best performance for this application. The Greedy and the Menu show very similar performances, whereas Probability is a bit worse than these two. The Breadth-First and the Depth-First are significantly worse than the

others.

7.6.9 Summary

Table 7.2 shows, for each application and for each strategy, the total number of event executions and the number of resets required by the strategy to discover all the states of the application (i.e., the data in this table corresponds to the last points of the plots presented). Based on these two numbers, the cost of discovering all the states are calculated and is shown next to them. The table also contains the optimal costs for each application.

Table 7.3 shows the statistical variations (due to shuffling of events) of the costs to discover all the states.

In all the cases, the model-based crawling strategies are significantly better than the Breadth-First and the Depth-First. It can be seen that the Breadth-First strategy uses significantly more resets and the Depth-First strategy executes significantly more events than the others.

	Bebop			Elfinder		
	Events	Resets	Cost	Events	Resets	Cost
Depth-First	13,188,873	1	13,188,875	1,998,329	194	2,000,559
Breadth-First	933,592	8,727	951,047	115,702	7,033	186,028
Greedy	806,315	27	806,369	67,187	190	69,085
Hypercube	794,102	27	794,156	67,453	195	69,402
Menu	32,604	1	32,606	48,438	203	50,466
Probability	27,447	27	27,501	49,731	134	51,076
Optimal	1,799	1	1,781	1,359	1	1,369

	FileTree			Periodic Table			Clipmarks		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	100,589	1	100,591	967,724	53	968,148	18,971	45	19,776
Breadth-First	24,028	1,575	27,178	28,898	7,506	88,947	12,669	902	28,906
Greedy	17,107	13	17,133	29,622	53	30,046	11,024	20	11,390
Hypercube	15,719	13	15,745	29,622	53	30,046	10,932	15	11,199
Menu	256	1	258	16,201	20	16,363	4,615	29	5,134
Probability	804	1	806	28,969	2	28,985	10,862	25	11,315
Optimal	213	1	215	239	1	247	128	2	164

	TestRIA			Altoro Mutual			Hypercube10D		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	1,219	1	1,221	7,496	20	7,536	23,033	4,090	35,303
Breadth-First	1,106	54	1,214	1,939	333	2,604	28,070	5,111	43,403
Greedy	900	1	902	1,872	19	1,909	7,269	1,002	10,275
Hypercube*	889	1	891	905	26	957	2,078	252	2,834
Menu	98	1	100	101	4	110	7,258	1,000	10,258
Probability	115	1	117	179	7	193	7,251	996	10,240
Optimal	57	1	59	72	1	74	1,646	252	2,402

Table 7.2: The number of events executed, the number of resets used, and the cost (as defined in Section 7.2) for discovering all the states. The numbers are rounded to the nearest integer to increase readability.

* For Hypercube10D, the Hypercube strategy executes more events than the optimal since our implementation of the Hypercube strategy keeps exploring more events when the end of an MCD chain is reached, rather than immediately resetting and following another MCD chain. Otherwise, these numbers would be the same as explained in Section 4.4.5.

	Bebop					Elfinder				
	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.
Depth-First	13,188,875	432,900	86,580	12,324,013	13,731,041	2,000,559	82,326	16,465	1,879,525	2,152,368
Breadth-First	951,047	518	104	950,233	952,196	186,028	2,363	473	181,149	190,040
Greedy	806,369	3,731	746	794,358	811,869	69,085	538	108	67,910	69,951
Hypercube	794,156	2,111	422	790,811	797,758	69,402	511	102	67,790	70,147
Menu	32,606	481	96	31,085	33,131	50,466	18,229	3,646	32,828	86,650
Probability	27,501	6,300	1,260	23,036	54,602	51,076	5,682	1,136	41,382	59,507

	FileTree					Periodic Table					Clipmarks				
	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.
Depth-First	100,591	9,043	1,809	80,436	114,485	968,148	118,756	23,751	723,704	1,156,163	19,776	912	182	18,252	22,407
Breadth-First	27,178	2,228	446	24,540	29,317	88,947	236	47	88,750	89,685	28,906	447	89	27,987	29,530
Greedy	17,133	271	54	16,678	17,829	30,046	327	65	29,584	30,636	11,390	338	68	10,847	11,839
Hypercube	15,745	423	85	14,991	16,302	30,046	327	65	29,584	30,636	11,199	241	48	10,698	11,491
Menu	258	3	1	252	264	16,363	7,261	1,452	2,571	26,262	5,134	3,261	652	3,229	11,920
Probability	806	152	30	614	1,209	28,985	117	23	28,820	29,271	11,315	617	123	9,315	11,708

	TestRIA					Altoro Mutual					Hypercube10D				
	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.
Depth-First	1,221	137	27	890	1,506	7,536	1,057	211	6,063	9,679	35,303	0	0	35,303	35,303
Breadth-First	1,214	19	4	1,178	1,246	2,604	324	65	2,097	3,023	43,403	3	1	43,391	43,403
Greedy	902	19	4	872	940	1,909	38	8	1,815	1,972	10,275	1,355	271	8,370	12,441
Hypercube	891	18	4	856	923	957	48	10	856	1,032	2,834	3	1	2,825	2,838
Menu	100	4	1	93	107	110	6	1	99	119	10,258	1,348	270	8,370	12,441
Probability	117	8	2	99	133	193	8	2	181	213	10,198	285	87	9,803	11,071

Table 7.3: Statistics for the costs to discover all the states. The numbers are rounded to the nearest integer to increase readability.

7.7 Total Cost of Crawling

In Table 7.4, we present the costs for completing the crawling of the application (i.e., the costs when crawling terminates). Table 7.5 shows the statistical variations (due to shuffling of events) of the costs to complete the crawl.

The results show that model-based crawling strategies have significantly better performance than the Breadth-First and the Depth-First strategies since all model-based crawling algorithms try to reduce the length of transfer sequences. The Greedy strategy also shows a good performance similar to model-based strategies.

	Bebop			Elfinder		
	Events	Resets	Cost	Events	Resets	Cost
Depth-First	13,386,210	27	13,386,264	2,000,490	209	2,002,580
Breadth-First	943,001	8732	960,466	121,908	7,133	193,242
Greedy	826,914	27	826,968	68,684	209	70,774
Hypercube	816,142	27	816,196	68,539	209	70,629
Menu	814,220	27	814,274	77,846	223	80,078
Probability	816,922	27	816,976	68,568	233	70,902

	FileTree			Periodic Table			Clipmarks		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	101,409	13	101,435	968,027	236	969,915	19,246	72	20,550
Breadth-First	26,376	1,639	29,654	64,853	14,634	181,921	15,362	933	32,154
Greedy	20,763	13	20,789	29,925	236	31,813	11,397	56	12,402
Hypercube	19,860	13	19,886	29,925	236	31,813	11,351	56	12,357
Menu	19,709	13	19,735	37,606	236	39,494	11,716	70	12,985
Probability	19,341	13	19,367	29,584	236	31,472	11,458	61	12,561

	TestRIA			Altoro Mutual			Hypercube10D		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	1,378	1	1,381	7,513	34	7,581	23,050	4,098	35,344
Breadth-First	1,217	55	1,327	3,075	334	3,742	28,160	5,120	43,520
Greedy	1,001	1	1,003	2,509	34	2,577	8,866	1,260	12,646
Hypercube	996	1	998	2,483	34	2,552	8,860	1,260	12,640
Menu	973	1	975	2,463	35	2,533	8,866	1,260	12,646
Probability	973	1	975	2,451	35	2,521	9,327	1,340	13,348

Table 7.4: The number of events executed, the number of resets used, and the cost (as defined in Section 7.2) when crawl terminates. The numbers are rounded to the nearest integer to increase readability.

	Bebop					Elfinder				
	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.
Depth-First	13,386,264	187,035	37,407	13,069,174	13,758,514	2,002,580	82,350	16,470	1,880,497	2,154,759
Breadth-First	960,466	98	20	960,307	960,599	193,242	1,603	321	190,049	196,540
Greedy	826,968	1,403	281	824,148	830,325	70,774	84	17	70,629	70,977
Hypercube	816,196	39	8	816,115	816,270	70,629	67	13	70,452	70,753
Menu	814,274	31	6	814,211	814,323	80,078	3,847	769	76,968	88,880
Probability	816,976	167	33	816,623	817,255	70,902	584	117	70,038	72,168

	FileTree					Periodic Table					Clipmarks				
	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.
Depth-First	101,435	9,232	1,846	81,452	115,455	969,915	118,779	23,756	725,471	1,157,849	20,550	900	180	19,127	23,208
Breadth-First	29,654	14	3	29,624	29,689	181,921	9	2	181,906	181,936	32,154	149	30	31,839	32,525
Greedy	20,789	64	13	20,658	20,923	31,813	13	3	31,783	31,849	12,402	14	3	12,365	12,419
Hypercube	19,886	21	4	19,850	19,953	31,813	14	3	31,783	31,849	12,357	9	2	12,336	12,372
Menu	19,735	33	7	19,664	19,780	39,494	2,693	539	34,607	43,112	12,985	166	33	12,757	13,422
Probability	19,367	40	8	19,300	19,443	31,472	75	15	31,387	31,609	12,561	62	12	12,462	12,736

	TestRIA					Altoro Mutual					Hypercube10D				
	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.	Mean	Std. Dev.	Std. Err.	Min.	Max.
Depth-First	1,381	111	22	1,248	1,661	7,581	1,060	212	6,080	9,757	35,344	0	0	35,344	35,344
Breadth-First	1,327	9	2	1,309	1,343	3,742	28	6	3,701	3,803	43,520	3	1	43,508	43,520
Greedy	1,003	7	1	989	1,015	2,577	14	3	2,543	2,605	12,646	2	0	12,642	12,651
Hypercube	998	6	1	989	1,018	2,552	13	3	2,522	2,570	12,640	0	0	12,640	12,640
Menu	975	2	0	971	980	2,533	11	2	2,514	2,562	12,646	2	0	12,642	12,651
Probability	975	1	0	973	977	2,521	7	1	2,506	2,535	13,378	185	37	12,994	13,904

Table 7.5: Statistics for the costs when the crawl terminates. The numbers are rounded to the nearest integer to increase readability.

7.8 Time Measurements

The results that are presented in the previous sections are obtained by simulating the strategies on the application models as we have explained in Section 7.5. In this section, we present the time measurements obtained by actually crawling each application with each strategy once. The time measurements are presented as Hours:Minutes:Seconds. The seconds are rounded to the nearest integer to increase readability; a time measurement of 00:00:00 means that the time spent is less than 0.5 seconds.

7.8.1 State Discovery and Complete Crawl Times

For each application, Tables 7.6-7.13 contain the cost and the time required by each strategy to discover all the states of the application, as well as to complete the crawl of the application.

As expected, the time measurements reflect the fact that strategies which execute less events and less resets often require proportionately less time. The major source of difference between the time measurements and our cost measurements is our simplifying assumption that each event execution takes the same average time. As a result of this simplification, for Clipmarks our measured value for cost of reset is a bit higher than the real time delays incurred by the resets during crawling. This is why the Breadth-First has more cost than Depth-First (since Breadth-First uses many resets), but actually Breadth-First finishes earlier than Depth-First. Except for this case, the cost measurements are in line with the time measurements.

It can be seen that an efficient strategy can discover the states much earlier, even if the crawl takes a long time. For example, even though it takes 8 hours for Probability to completely crawl Bebop, all the states are discovered in the first 13 minutes of the crawl (as it can be seen in Table 7.6). Similarly Menu discovers all the states in this application in the first 19 minutes while the crawl completes in 13 hours.

For Bebop and Elfinder, the Menu strategy completes the crawl much later than the other strategies that require a cost comparable to the Menu (the Menu strategy still discovers the states in a comparable time to the Probability and earlier than the other strategies during the crawl though). This excess time is used for calculating a Chinese Postman Path (CPP) which is used by the Menu strategy in its transition exploration phase. Since these two applications have relatively larger models, it takes more time to calculate a Chinese Postman Path for these applications. This is partly because, the current code used to solve CPP is not implemented very efficiently[65]. The time required

by the Menu to complete the crawl might be reduced with a better implementation of the CPP algorithm[10].

In all applications, the Breadth-First and the Depth-First require more time to discover all the states than the other strategies. The same is true for completing the crawl with the exception of Menu on Bebop and Elfinder.

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	12,542,684	1	12,542,686	93:42:14	13,006,646	27	13,006,700	97:05:22
Breadth-First	933,696	8,681	951,058	09:00:31	943,035	8,686	960,407	09:04:57
Greedy	803,966	27	804,020	08:17:07	827,830	27	827,884	08:32:04
Hypercube	796,356	27	796,410	08:10:36	816,187	27	816,241	08:22:54
Menu	32,960	1	32,962	00:18:49	814,148	27	814,202	12:48:55
Probability	23,451	27	23,505	00:12:50	817,044	27	817,098	07:54:03

Table 7.6: Costs and Time Measurements for Bebop. The time format is Hours:Minutes:Seconds.

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	1,999,241	199	2,001,231	55:15:56	2,000,239	209	2,002,329	55:17:30
Breadth-First	116,562	7,178	188,342	04:42:47	121,735	7,258	194,315	04:53:40
Greedy	66,806	185	68,656	02:18:56	68,534	209	70,624	02:21:47
Hypercube	66,886	184	68,726	02:19:29	68,481	209	70,571	02:22:05
Menu	41,915	201	43,925	02:01:29	75,346	213	77,476	08:23:26
Probability	39,526	200	41,526	01:09:14	68,152	278	70,932	02:03:43

Table 7.7: Costs and Time Measurements for Elfinder. The time format is Hours:Minutes:Seconds.

[10]Some suggestions for optimizations are provided in [65].

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	105,343	1	105,345	01:43:06	105,645	13	105,671	01:43:28
Breadth-First	21,654	1495	24,644	00:38:29	26,370	1,638	29,646	00:43:05
Greedy	17,134	13	17,160	00:29:57	20,755	13	20,781	00:33:42
Hypercube	16,243	13	16,269	00:29:10	19,887	13	19,913	00:33:02
Menu	253	1	253	00:00:15	19,741	13	19,767	00:33:40
Probability	779	1	779	00:00:36	19,360	13	19,386	00:31:37

Table 7.8: Costs and Time Measurements for FileTree. The time format is Hours:Minutes:Seconds.

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	860,312	7	860,368	05:37:45	860,661	236	862,549	05:38:02
Breadth-First	28,826	7,495	88,786	00:21:48	64,853	14,634	181,925	00:42:19
Greedy	29,574	7	29,630	00:14:44	29,923	236	31,811	00:15:06
Hypercube	29,574	7	29,630	00:15:20	29,923	236	31,811	00:15:42
Menu	16,846	4	16,878	00:08:49	41,082	236	42,970	00:20:06
Probability	28,904	2	28,920	00:14:15	29,525	236	31,413	00:14:42

Table 7.9: Costs and Time Measurements for Periodic Table. The time format is Hours:Minutes:Seconds.

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	18,617	46	19,445	00:14:44	18,799	71	20,077	00:14:52
Breadth-First	12,608	900	28,808	00:07:17	15,327	925	31,977	00:08:14
Greedy	10,748	7	10,874	00:04:11	11,390	55	12,380	00:04:28
Hypercube	10,732	7	10,858	00:04:12	11,363	56	12,371	00:04:29
Menu	3,106	9	3,268	00:01:55	11,623	69	12,865	00:04:22
Probability	11,114	33	11,708	00:04:32	11,492	62	12,608	00:04:39

Table 7.10: Costs and Time Measurements for Clipmarks. The time format is Hours:Minutes:Seconds.

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	1302	1	1,304	00:00:27	1,460	1	1,462	00:00:34
Breadth-First	1097	54	1,205	00:00:25	1,229	55	1,339	00:00:27
Greedy	884	1	886	00:00:20	1,006	1	1,008	00:00:21
Hypercube	882	1	884	00:00:19	995	1	997	00:00:21
Menu	96	1	98	00:00:03	974	1	976	00:00:21
Probability	127	1	129	00:00:04	973	1	975	00:00:21

Table 7.11: Costs and Time Measurements for TestRIA. The time format is Hours:Minutes:Seconds.

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	6,790	25	6,840	00:02:37	6,802	34	6,870	00:02:38
Breadth-First	1,635	352	2,339	00:00:54	3,076	353	3,782	00:01:25
Greedy	1,842	25	1,892	00:00:44	2,521	34	2,589	00:01:00
Hypercube	906	32	970	00:00:24	2,490	34	2,558	00:01:00
Menu	104	7	118	00:00:04	2,468	34	2,536	00:01:00
Probability	172	7	186	00:00:06	2,452	35	2,522	00:00:59

Table 7.12: Costs and Time Measurements for Altoro Mutual. The time format is Hours:Minutes:Seconds.

	Discovering All States				Complete Crawl			
	Events	Resets	Cost	Time	Events	Resets	Cost	Time
Depth-First	23,033	4,090	31,213	00:11:59	23,050	4,098	35,344	00:12:00
Breadth-First	28,070	5,111	38,292	00:15:05	28,160	5,120	43,520	00:15:07
Greedy	7,093	972	9,037	00:03:34	8,865	1,260	12,645	00:04:28
Hypercube	2,077	252	2,581	00:01:03	8,860	1,260	12,640	00:04:19
Menu	7,093	972	9,037	00:03:33	8,865	1,260	12,645	00:04:24
Probability	7,335	1,012	9,359	00:03:44	9,434	1,364	13,526	00:04:46

Table 7.13: Costs and Time Measurements for Hypercube10D. The time format is Hours:Minutes:Seconds.

7.8.2 Distributions of the Complete Crawl Times

Tables 7.14-7.21 show for each strategy how the complete crawl time is distributed among the following operations:

- **Strategy** shows the total time spent by the crawling strategy. This time is spent for deciding which event to explore next, calculating the transfer sequence to the state where the next event will be explored, and updating the model that is being extracted.
- **Event Execution** shows the total time required to execute events. This includes the time executing JavaScript code, modifying the DOM accordingly, and the time delays for AJAX calls.
- **Reset** shows the total time spent for resets.
- **DOM ID** shows the total time spent by the DOM Equivalence algorithm which produces an identifier for the reached DOM after each event exploration. This algorithm is not executed for the DOMs visited while executing a transfer sequence.
- **Event ID** shows the total time required to detect the events on a DOM, and producing identifiers for these events. This is done after every event execution (which includes the events executed in a transfer sequence) since we need to locate an event first in order to execute it.

These tables also show the ratio of the time taken by each operation to the total crawl time as a percentage.

Unsurprisingly, the results show that the crawling time is dominated by event executions and resets. For all strategies (except for Menu), event executions and resets combined takes more than 75% of the crawl time in 6 out of the 8 applications. For the remaining 2 applications (Periodic Table and Clipmarks), event executions and resets take more than 60% of the crawl time. Event ID calculation is the most time consuming operation after event executions and resets since event identifiers are calculated after each event execution. DOM ID calculation usually does not take a significant amount of time (usually less than 3%); however, in Clipmarks and Periodic Table it takes more time to calculate DOM IDs than the other applications.

Except for Menu, the strategies themselves take insignificant amount of time during the crawl (often less than 1%). Because of the Chinese Postman Path calculation in the

transition exploration phase, the Menu strategy needs more time than the other strategies. The differences are more significant for Bebop and Elfinder since these applications have relatively larger models.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:03:44	0.06%	63:00:39	64.90%	00:00:02	0.0005%	00:05:08	0.09%	33:55:49	34.95%	97:05:22
Breadth-First	00:03:50	0.70%	06:52:53	75.76%	00:08:41	1.59%	00:05:16	0.97%	01:54:17	20.97%	09:04:57
Greedy	00:03:18	0.64%	06:30:31	76.26%	00:00:02	0.01%	00:05:31	1.08%	01:52:42	22.01%	08:32:04
Hypercube	00:04:33	0.90%	06:22:31	76.06%	00:00:02	0.01%	00:05:29	1.09%	01:50:19	21.94%	08:22:54
Menu	04:58:40	38.84%	06:03:28	47.27%	00:00:02	0.004%	00:05:10	0.67%	01:41:36	13.21%	12:48:55
Probability	00:08:10	1.72%	06:04:08	76.81%	00:00:02	0.01%	00:05:12	1.10%	01:36:31	20.36%	07:54:03

Table 7.14: Distribution of the Times to Complete the Crawl for Bebop. The time format is Hours:Minutes:Seconds.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:00:13	0.01%	52:37:43	95.27%	00:03:18	0.10%	00:00:34	0.02%	02:32:41	4.61%	55:17:30
Breadth-First	00:00:10	0.06%	03:52:33	79.19%	00:54:11	18.45%	00:00:34	0.19%	00:06:12	2.11%	04:53:40
Greedy	00:00:12	0.14%	02:14:20	94.74%	00:01:41	1.19%	00:00:34	0.40%	00:05:00	3.53%	02:21:47
Hypercube	00:00:14	0.17%	02:14:33	94.70%	00:01:44	1.21%	00:00:34	0.40%	00:05:01	3.53%	02:22:05
Menu	05:43:29	68.23%	02:32:03	30.20%	00:01:45	0.35%	00:00:33	0.11%	00:05:36	1.11%	08:23:26
Probability	00:00:50	0.67%	01:55:44	93.54%	00:02:19	1.88%	00:00:33	0.45%	00:04:17	3.46%	02:03:43

Table 7.15: Distribution of the Times to Complete the Crawl for Elfinder. The time format is Hours:Minutes:Seconds.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:00:01	0.02%	01:32:22	89.27%	00:00:03	0.05%	00:00:26	0.42%	00:10:36	10.24%	01:43:28
Breadth-First	00:00:01	0.05%	00:38:28	89.26%	00:02:55	6.79%	00:00:27	1.06%	00:01:13	2.84%	00:43:05
Greedy	00:00:01	0.06%	00:29:50	88.55%	00:00:03	0.17%	00:00:26	1.29%	00:03:21	9.93%	00:33:42
Hypercube	00:00:02	0.09%	00:29:00	87.76%	00:00:04	0.19%	00:00:27	1.35%	00:03:30	10.61%	00:33:02
Menu	00:00:13	0.65%	00:29:12	86.71%	00:00:03	0.17%	00:00:26	1.28%	00:03:46	11.18%	00:33:40
Probability	00:00:03	0.14%	00:28:07	88.90%	00:00:04	0.19%	00:00:26	1.38%	00:02:58	9.39%	00:31:37

Table 7.16: Distribution of the Times to Complete the Crawl for FileTree. The time format is Hours:Minutes:Seconds.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:00:09	0.05%	04:22:43	77.72%	00:00:20	0.10%	00:03:10	0.94%	01:11:41	21.20%	05:38:02
Breadth-First	00:00:10	0.38%	00:11:31	27.21%	00:19:16	45.55%	00:04:35	10.82%	00:06:47	16.04%	00:42:19
Greedy	00:00:13	1.47%	00:08:48	58.25%	00:00:20	2.15%	00:03:17	21.70%	00:02:29	16.42%	00:15:06
Hypercube	00:00:13	1.40%	00:09:19	59.36%	00:00:20	2.10%	00:03:14	20.61%	00:02:36	16.53%	00:15:42
Menu	00:00:58	4.83%	00:11:40	58.02%	00:00:20	1.65%	00:03:24	16.87%	00:03:45	18.63%	00:20:06
Probability	00:00:19	2.20%	00:08:27	57.50%	00:00:20	2.21%	00:03:12	21.74%	00:02:24	16.35%	00:14:42

Table 7.17: Distribution of the Times to Complete the Crawl for Periodic Table. The time format is Hours:Minutes:Seconds.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:00:04	0.48%	00:12:25	83.48%	00:00:13	1.43%	00:00:39	4.32%	00:01:32	10.29%	00:14:52
Breadth-First	00:00:04	0.87%	00:03:46	45.70%	00:02:27	29.66%	00:00:41	8.31%	00:01:16	15.47%	00:08:14
Greedy	00:00:05	1.68%	00:02:39	59.17%	00:00:10	3.69%	00:00:39	14.51%	00:00:56	20.95%	00:04:28
Hypercube	00:00:05	1.74%	00:02:38	58.81%	00:00:10	3.76%	00:00:39	14.64%	00:00:57	21.04%	00:04:29
Menu	00:00:13	5.03%	00:02:19	52.99%	00:00:12	4.64%	00:00:39	15.05%	00:00:58	22.29%	00:04:22
Probability	00:00:06	2.27%	00:02:44	58.98%	00:00:11	4.06%	00:00:39	14.11%	00:00:57	20.58%	00:04:39

Table 7.18: Distribution of the Times to Complete the Crawl for Clipmarks. The time format is Hours:Minutes:Seconds.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:00:00	0.10%	00:00:32	94.47%	00:00:01	2.76%	00:00:00	0.73%	00:00:01	1.94%	00:00:34
Breadth-First	00:00:00	0.13%	00:00:23	84.95%	00:00:03	12.13%	00:00:00	0.89%	00:00:01	1.91%	00:00:27
Greedy	00:00:00	0.11%	00:00:20	92.25%	00:00:01	4.32%	00:00:00	1.22%	00:00:00	2.10%	00:00:21
Hypercube	00:00:00	0.21%	00:00:19	92.15%	00:00:01	4.48%	00:00:00	1.07%	00:00:00	2.09%	00:00:21
Menu	00:00:00	0.71%	00:00:19	91.74%	00:00:01	4.40%	00:00:00	1.06%	00:00:00	2.08%	00:00:21
Probability	00:00:00	0.26%	00:00:19	92.13%	00:00:01	4.47%	00:00:00	1.19%	00:00:00	1.95%	00:00:21

Table 7.19: Distribution of the Times to Complete the Crawl for TestRIA. The time format is Hours:Minutes:Seconds.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:00:00	0.08%	00:02:27	93.58%	00:00:02	1.37%	00:00:02	0.99%	00:00:06	3.99%	00:02:38
Breadth-First	00:00:00	0.13%	00:01:05	76.30%	00:00:15	17.30%	00:00:02	2.41%	00:00:03	3.86%	00:01:25
Greedy	00:00:00	0.20%	00:00:53	89.45%	00:00:02	3.71%	00:00:02	2.68%	00:00:02	3.96%	00:01:00
Hypercube	00:00:00	0.28%	00:00:54	89.48%	00:00:02	3.49%	00:00:02	2.70%	00:00:02	4.05%	00:01:00
Menu	00:00:00	0.74%	00:00:53	88.96%	00:00:02	3.73%	00:00:02	2.62%	00:00:02	3.95%	00:01:00
Probability	00:00:00	0.36%	00:00:52	89.26%	00:00:02	3.66%	00:00:02	2.73%	00:00:02	3.98%	00:00:59

Table 7.20: Distribution of the Times to Complete the Crawl for Altoro Mutual. The time format is Hours:Minutes:Seconds.

	Strategy		Event Execution		Reset		DOM ID		Event ID		Total
	Time	%	Time	%	Time	%	Time	%	Time	%	Time
Depth-First	00:00:01	0.11%	00:06:48	56.66%	00:04:58	41.33%	00:00:03	0.42%	00:00:11	1.48%	00:12:00
Breadth-First	00:00:01	0.13%	00:08:24	55.56%	00:06:26	42.51%	00:00:04	0.39%	00:00:13	1.42%	00:15:07
Greedy	00:00:01	0.33%	00:02:36	58.26%	00:01:45	39.22%	00:00:02	0.78%	00:00:04	1.41%	00:04:28
Hypercube	00:00:01	0.22%	00:02:34	59.56%	00:01:38	37.96%	00:00:02	0.79%	00:00:04	1.47%	00:04:19
Menu	00:00:02	0.59%	00:02:39	60.52%	00:01:37	36.68%	00:00:02	0.78%	00:00:04	1.43%	00:04:24
Probability	00:00:02	0.74%	00:02:49	59.10%	00:01:49	38.02%	00:00:02	0.75%	00:00:04	1.39%	00:04:46

Table 7.21: Distribution of the Times to Complete the Crawl for Hypercube10D. The time format is Hours:Minutes:Seconds.

7.9 Conclusion

We conducted experiments on five real AJAX-based RIAs and three test applications. We evaluated strategy efficiency in terms of the number of events executed and the resets used to discover all the states in an application. The results show that model-based crawling strategies and the Greedy strategy discover the states by executing significantly fewer events and resets than the Depth-First and the Breadth-First. The Breadth-First strategy uses significantly more resets, and the Depth-First strategy executes significantly more events than the others. The performance of the Hypercube strategy is similar to the Greedy strategy when the application does not follow the Hypercube meta-model. In most cases, the Probability and the Menu show comparable performances, and they are the most efficient of all.

We also presented the time measurements. Time measurements show that event executions and resets are normally the operations that dominate the time spent during the crawl. The strategies that use fewer events and resets to discover the states also

require less time. Thus, the Probability and the Menu strategy discover the states much earlier than the others.

Except for the Menu strategy, the time it takes to complete the crawl is also in line with the number of event executions and resets. Since the Menu strategy calculates a Chinese Postman Path (CPP) for its transition exploration phase, the Menu strategy requires more time to complete the crawl than the other strategies that execute comparable number of events and resets. This is partly because of the inefficiency of the code used to find CPP. The time for Menu might be improved by optimizing the implementation of the CPP algorithm.

Chapter 8

Conclusion and Future Directions

8.1 Conclusion

With RIA technologies, the web applications have become more interactive and responsive. Although this is an improvement in terms of user-friendliness, these technologies come at the cost of not being able to use the crawling techniques established so far. It is important to regain the ability to crawl RIAs to be able to search their content and build their models for various purposes such as reverse-engineering, detecting security vulnerabilities, assessing usability, and applying model-based testing techniques.

Compared with the research on crawling traditional applications, where solutions to many different problems are proposed, the research on crawling RIAs is very recent. Even the basic problem of efficiently discovering the pages in a RIA has not been addressed completely. In our definition, an efficient strategy is the one that discovers the states of the application as soon as possible. Discovering the states sooner is important because more information will be available for analysis earlier, even if the crawl takes very long time to complete. Although the majority of the existing research on crawling RIAs use the standard crawling strategies, Breadth-First and Depth-First, these are not efficient for crawling RIAs.

This thesis expands the research on crawling RIAs by providing crawling strategies that are more efficient than Breadth-First and Depth-First. For this purpose, we follow a general approach called model-based crawling. In model-based crawling, we design crawling strategies that aim at discovering the states of the application as early as possible during the crawl based on some anticipations about the behavior of the application. In this thesis, we have presented two model based crawling strategies: an improved version

of the first model-based strategy, the Hypercube strategy, and the Probability strategy.

To evaluate the performances of these strategies with the existing ones, a prototype crawler for AJAX-based applications has been developed. We presented an experimental study conducted on five real AJAX-based applications and three test applications. The results show that model-based crawling strategies are more efficient than the standard crawling strategies in all the cases. We have seen that the Hypercube strategy performs similar to a Greedy strategy when the applications does not follow the model anticipated by the Hypercube strategy, otherwise the Hypercube strategy is optimal. However, the anticipations of the Hypercube strategy are not realized by most of the real applications, so it is hard to find real examples where the Hypercube strategy uses its full potential. On the other hand, the Probability strategy is more relaxed and is more efficient than the Hypercube strategy on real RIAs. The performance of the Probability strategy is often comparable to the Menu strategy (another model-based strategy), but the Probability strategy is easier to implement than the Menu strategy.

We conclude the thesis with some future directions.

8.1.1 Adaptive Model-based Crawling

One important aspect of model-based crawling is to decide on a meta-model for which the crawling strategy will be optimized. However, it is often difficult to predict a good meta-model for an arbitrary application before crawling. A possible solution to this problem might be using an adaptive model-based crawling approach. Instead of fixing a meta-model before crawling, the crawler could start exploring the application using a strategy that does not have strict assumptions and that is known to produce good results, such as the Probability strategy. Once some initial information is collected using this strategy, the partially extracted model could be analyzed by the crawler, and a model-based strategy that would suit the application could be chosen.

This idea of dynamically choosing the meta-model can even be developed further, so that a suitable meta-model could be constructed during the crawl. This would be possible when the model of the application has some repeating patterns. For example, we might detect that the instances of the same subgraph repeats itself in the partially extracted model (as an example, we can think of a large application that uses the same navigational pattern to present different content, like a web-based product catalog). In that case, it could even be possible to generate an optimal strategy for such subgraphs. Whenever the crawler can predict that some portion of the application is likely to follow

this same structure, we can apply this dynamically generated, optimized strategy for exploring that portion.

8.1.2 State-Space Explosion

RIAs tend to have large number of states. No matter how efficient the strategies we design are, eventually all strategies are all susceptible to the state explosion problem unless additional steps are taken. This is the reason why most of the time we had to crawl small instances of the subject applications. This problem is often caused by considering the whole page to define the states even though the page might consist of parts that can be interacted independently. For example, each widget in a widget-based application, like the one shown in Figure 8.1, can be interacted independent of each other. Currently, every different combination of the contents of such independent parts will be considered as a new state. However, the majority of such states will not contain any new information. Being able to detect such independent components and crawling each such component separately can be a possible way to address this problem.

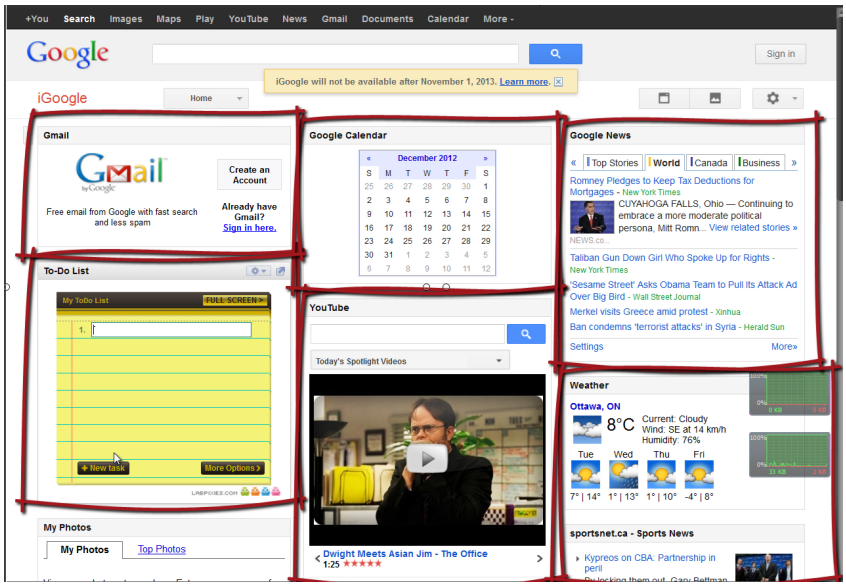

Figure 8.1: A web page with multiple widgets

8.1.3 Greater Diversity

For a large RIA, it is not feasible to wait until crawling finishes to analyze the pages discovered. The analysis of the discovered pages usually takes place while crawling still

continues. Rather than exploring one part of the application exhaustively and keep discovering new but very similar pages, we would like discover dissimilar pages as much as possible earlier on during the crawl. For example, consider a web page has a long list of events where each event leads to a similar page. It is not reasonable to explore each of these events first, before trying something outside this list. This is true, especially for testing, since the similar pages would probably have the same problems. It does not have much use to find thousand instances of the same problem when finding one of them would suffice to fix all the instances. For this reason, new techniques are needed that would diversify the crawling and provide a bird-eye-view of the application as soon as possible. To this end, crawling strategies may benefit from algorithms that will help detecting similar pages, and events with similar functionality.

8.1.4 Relaxing the Determinism Assumption

Another common limitation of the current RIA crawling approaches is the determinism assumption, that is, the expectation that an event will lead to the same state whenever it is executed from a given state. This is not very realistic since most real web applications may react differently at different times. Crawling strategies should be improved in order to handle such cases.

8.1.5 Distributed Crawling

Another promising research area is to crawl RIAs using multiple concurrently running processes to reduce crawling time. The existing distributed crawling techniques for traditional applications distribute the workload based on URLs. However, this would not be sufficient in the context of RIA crawling, so new distributed crawling algorithms are required for RIAs.

8.1.6 Mobile Applications

With the increasing popularity of mobile devices (smartphones, tablets etc.), there is a growing number of applications developed specifically for such devices, called mobile applications. An emerging research area aims at building models of mobile applications for testing[8, 9, 35]. The crawling strategies that are designed for RIAs can easily be adapted to build models of mobile applications. This is because, the crawling strategies presented in this thesis do not rely on a particular technology. The underlying technology

affects the algorithms to identity the states and the events, which are given to the crawling strategy as inputs.

Appendix A

Experimental Results for Alternative Versions of Probability Strategy

A.1 Introduction

In this appendix, we present some further experimental results obtained using the alternative versions of the Probability strategy (the alternative versions are introduced in Section 5.6). In Section A.2, we present the results for alternative algorithms to choose the next state. In Section A.3, we present the results for the alternative probability estimation methods and the aging technique. Section A.4 shows the results when different values are used as the initial probability.

For compactness of presentation, we summarize the results using box plots. A box plot is presented for each different version of the strategy. 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. 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.

A.2 Algorithms to Choose a State

In Figure A.1, we compare three different mechanisms to choose the state where the next event should be explored:

- **Default:** The default strategy as explained in Section 5.5.1 (where two states are compared by using the iterated probability of the state that requires a shorter transfer sequence).
- **Alternative:** The alternative strategy where the state that minimizes the expected cost to discover a state is chosen.
- **Simple:** The simple strategy where the state that has the maximum probability is chosen regardless of the transfer sequence required to reach the state.

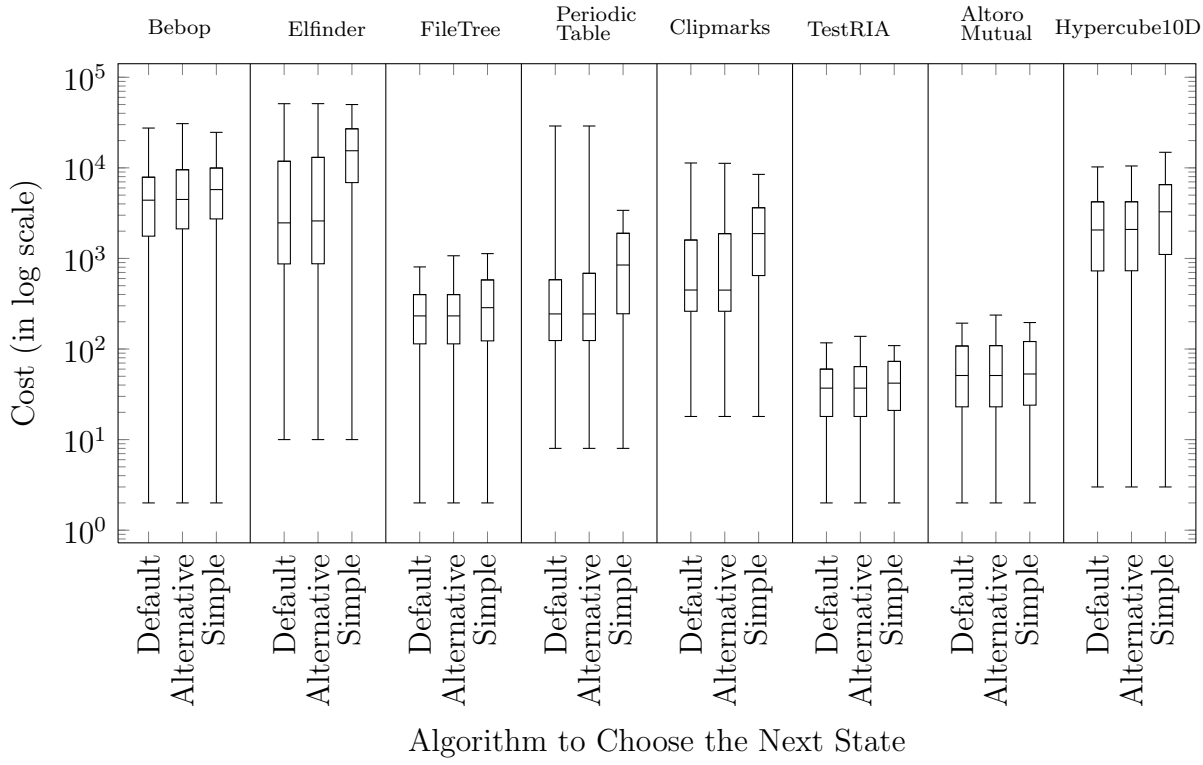

Figure A.1: State Discovery Cost for Different Algorithms to Choose the Next State (in log scale)

It can be seen that the default strategy is slightly better than the alternative version in the case of FileTree, TestRIA and Altoro Mutual. In the other cases, they are similar.

In all the cases, the default version perform better than the simple strategy in terms of the cost to discover 25%, 50% and 75% of the states. However, the simple strategy gets better towards the end and makes up for the difference. It even discovers all the states earlier than the default strategy in some of the cases. The most significant difference

for discovering all the states is in Periodic Table. For this application, the difference is caused only by the last state. We can see the results for the Periodic Table in more detail in Figure A.2. The default version (and also the alternative version) is significantly better than the simple version until the last state discovered. The reason is that in Periodic Table there is a state that can only be reached through the initial state and the initial state is not reachable from the other states unless a reset is used. The default and alternative versions only use a reset towards the end of the crawl. However, simple strategy uses a reset much earlier since it does not take the length of the transfer sequence into account.

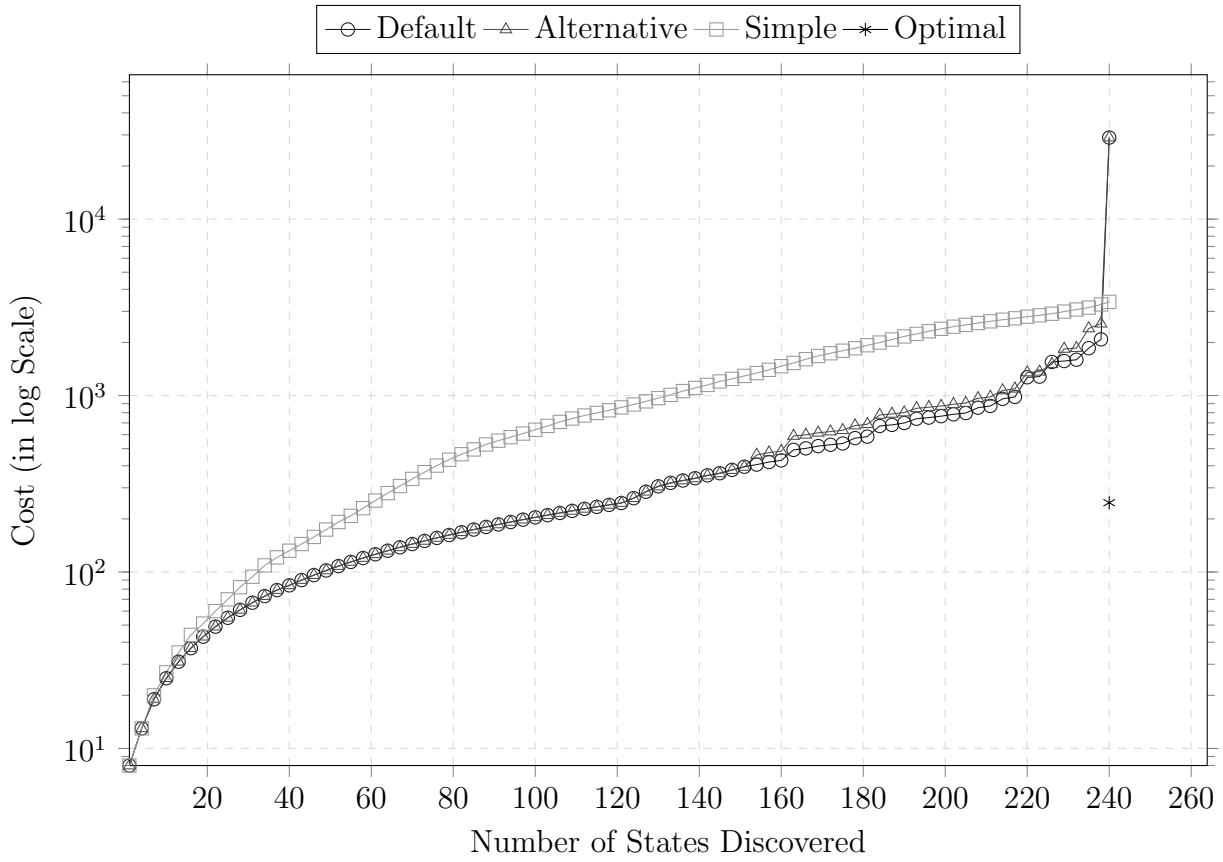

Figure A.2: State Discovery Costs for Periodic Table Using Different Ways to Choose a State (in log scale)

We can note that the methods that take the length of the transfer sequence into account when choosing a state, namely the default and alternative methods, discover the majority of the states much earlier than the simple algorithm in all the applications.

However, the simple method performs better towards the end. Between the default and alternative versions, the default one is slightly better.

A.3 Alternative Probability Estimations and Aging

Next, we compare the probability estimation techniques that give more importance to the recent explorations of an event (i.e., moving average techniques). These techniques are used together with the aging technique. At the end of the section, we present the results for alternative estimation of P_{avg}.

A.3.1 Moving Average Techniques and Aging

We have experimented with two different versions of moving average: Simple Moving Average (SMA) and Exponentially Weighted Moving Average (EWMA). For SMA, there is a parameter, w, specifying the window size for observations. That is, only the most recent w explorations are taken into account to estimate the probability of an event. In this study, we experimented with the following values for w: $w = 10, w = 50, w = 100$, and $w = \infty$. For EWMA, there is the "smoothing parameter", α, that specifies the importance given to recent explorations. We experimented with the values $\alpha = 0.1$, $\alpha = 0.01$, and $\alpha = 0$.

We used these moving average techniques in combination with the aging technique. The parameter for the aging technique is τ which specifies the threshold to boost an event's probability. That is, an event's probability is boosted when its age becomes larger than τ. We experimented with $\tau = 0.7, \tau = 0.8, \tau = 0.9$, and $\tau = 1$. The case when $w = \infty$ and $\tau = 1$ and the case when $\alpha = 0$ and $\tau = 1$ mean that every exploration of an event is taken into account when estimating its probability and the aging technique is not applied. Hence, these combinations represent the default strategy.

We present the results for each application separately. For each application, two sets of box plots are presented on the same page: the plots at the top of each page show the results for the SMA technique and the plots at the bottom show the results for the EWMA technique. The box plots are presented in linear scale except for Bebop and Periodic Table. For these applications logarithmic scale is used for better representation.

Bebop

Figure A.3 and Figure A.4 show the results for Bebop using SMA and EWMA, respectively. Both plots are in logarithmic scale. For this application, both moving average techniques perform similarly: taking as much explorations of an event as possible into account (large w or small α) gives better results. The best results are obtained when $w = \infty$ or $\alpha = 0$. The aging technique does not show any significant change in the results.

Elfinder

Figure A.5 and Figure A.6 show the results for Bebop using SMA and EWMA, respectively. In the case of SMA, using a limited window reduces the cost to discover 75% of the states. However, it does not perform so well for discovering the last quarter of the states: $w = \infty$ finds all the states faster. EWMA technique performs better than the SMA technique: EWMA is still better for discovering 75% of the states and it does not get worse in the last quarter like SMA. Thus, EWMA improves the default strategy. The aging technique does not have a significant effect on the results in both cases.

FileTree

Figure A.7 and Figure A.8 show the results for FileTree using SMA and EWMA, respectively. In this application, using a moving average technique does not seem to have an effect on the results. This may be explained by the fact that in this application, the first exploration of an event discovers a state, but the subsequent explorations of the event do not. Thus, the weight given to the recent explorations does not change the way strategy chooses the next event since there is not much variation in an event's possibility of discovering a state during the crawl. On the other hand, aging improves the results and the best results obtained when $\tau = 0.9$. This is because, aging causes an event's first exploration to be earlier: since initially each event's age is 1, an event that has not been explored yet gets the highest probability.

Periodic Table

Figures A.9 and A.10 show the results for Periodic Table (in logarithmic scale) using SMA and EWMA, respectively. The SMA technique does not have an observable effect on the results. But, EWMA with $\alpha = 0.1$ discovers the last state slightly earlier than the

default version. The aging technique reduces the cost of discovering 50% of the states. This is because, majority of the events (clicking on the elements in the table) lead to a new state in the first exploration and aging causes the first exploration of an event to happen earlier.

Clipmarks

Figure A.11 and Figure A.12 show the results for Clipmarks using SMA and EWMA, respectively. There is no improvement when a moving average technique is used, instead the results gets worse slightly. Using the aging technique seems to improve slightly the cost of discovering 75% of the states.

TestRIA

Figure A.13 and Figure A.14 show the results for TestRIA using SMA and EWMA, respectively. When SMA is used, the cases for $w = 50$ and $w = 100$ are identical to $w = \infty$ since there are at most 39 instances of an event in TestRIA. Using either moving average technique does not improve the results. In both cases, the aging technique slightly reduces the cost of discovering all the states, but 50% and 75% of the states are discovered significantly faster without aging.

Altoro Mutual

Figure A.15 and Figure A.16 show the results for Altoro Mutual using SMA and EWMA, respectively. When SMA is used, the cases for $w = 50$ and $w = 100$ are identical to $w = \infty$ since there are at most 45 instances of an event in Altoro Mutual. Using either moving average technique does not improve the results. In both cases, the aging technique slightly improves the results where the best results obtained when $\tau = 0.9$.

Hypercube10D

Figure A.17 and Figure A.18 show the results for Hypercube10D using SMA and EWMA, respectively. Using the moving average techniques improve the results to discover 50%, and 75% of the states. The best results are obtained with $w = 10$ in the case of SMA, and with $\alpha = 0.1$ in the case of EWMA. The aging technique has no significant effect on this application.

A.3.2 Using EWMA for Both Event Probabilities and $P_{\mathbf{avg}}$

Now, we present the results for the case when the EWMA technique is used to estimate P_{avg}, in addition to event probabilities. The same value for α is used to estimate P_{avg} and the event probabilities. The values we have experimented with are $\alpha = 0.1$, $\alpha = 0.01$, and $\alpha = 0$. $\alpha = 0$ is the case which corresponds to the default version of the strategy.

The box plots in Figure A.19 show the results for each application. For Elfinder, FileTree, Periodic Table and Clipmarks, this technique with $\alpha = 0.01$ is better than the default version ($\alpha = 0$). That is, the results for discovering the last quarter of the states improve and the performance up to that point is the same level as the default version. An exception is Bebop, where the default version discovers all the states faster, but the version with $\alpha = 0.01$ is better up to the point 75% of the states are discovered. For TestRIA, Altoro Mutual, and Hypercube10D, there is not any significant change.

A.4 Default Strategy with Different Initial Probabilities

For the default version of the strategy, the value of the initial probability is taken as 0.75. We have chosen 0.75 as the default value since it is usually better than the other values. Figure A.20 presents a comparison for the default strategy when the initial probability is taken as 0.25 and 0.5, in addition to the default value 0.75. We can see that the default value 0.75 is better than the other values for FileTree, Periodic Table, Altoro Mutual and for TestRIA. For Clipmarks, taking the initial probability as 0.25 is slightly faster up to the point where 75% of the states are discovered.

A.5 Conclusion

In this appendix, we provided some further experimental results for the Probability strategy. Although some of these techniques improved the default version for some applications, there is not a single version that gives the best performance in all the cases. In most cases, considering every exploration of an event for probability estimation gives fairly good results. Using the EWMA-based P_{avg} with $\alpha = 0.01$ improves the performance of the strategy for discovering the last quarter of the states in some cases. The aging does not seem to have much impact, except for the applications where the majority of the states can be discovered with the first execution of each event. The default value for the

initial probability (0.75) gives better results than the other values we have experimented with.

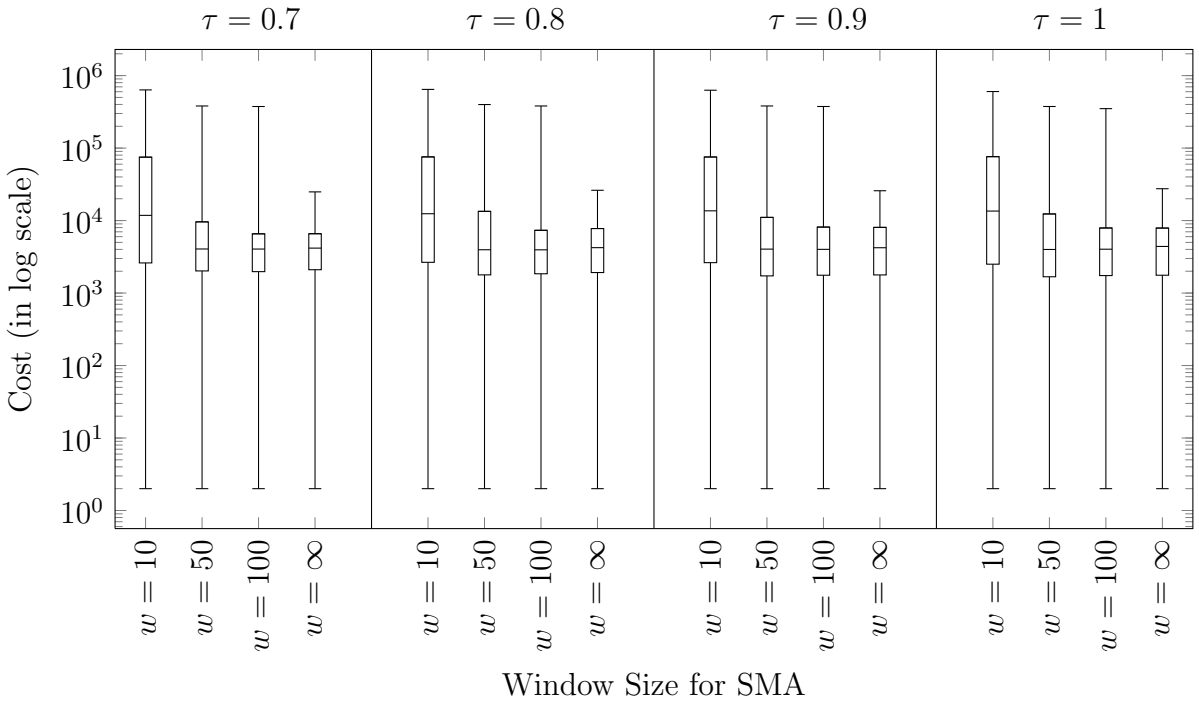

Figure A.3: State Discovery Costs using SMA and Aging for Bebop (in log scale)

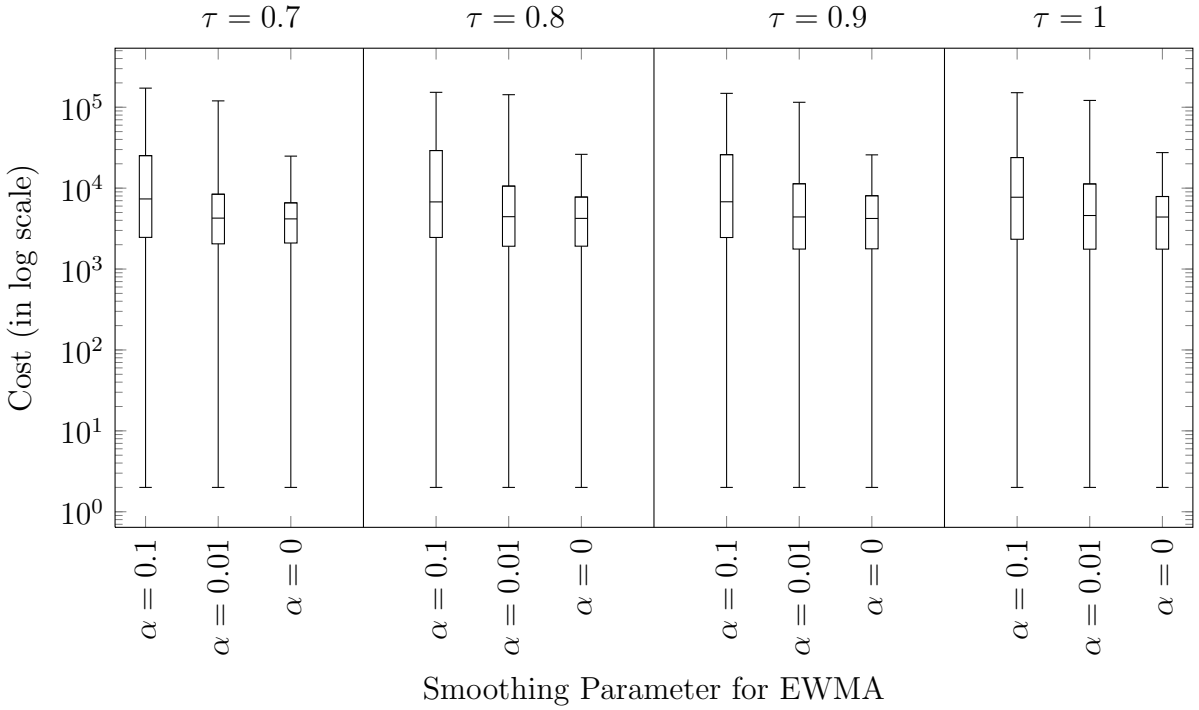

Figure A.4: State Discovery Costs using EWMA and Aging for Bebop (in log scale)

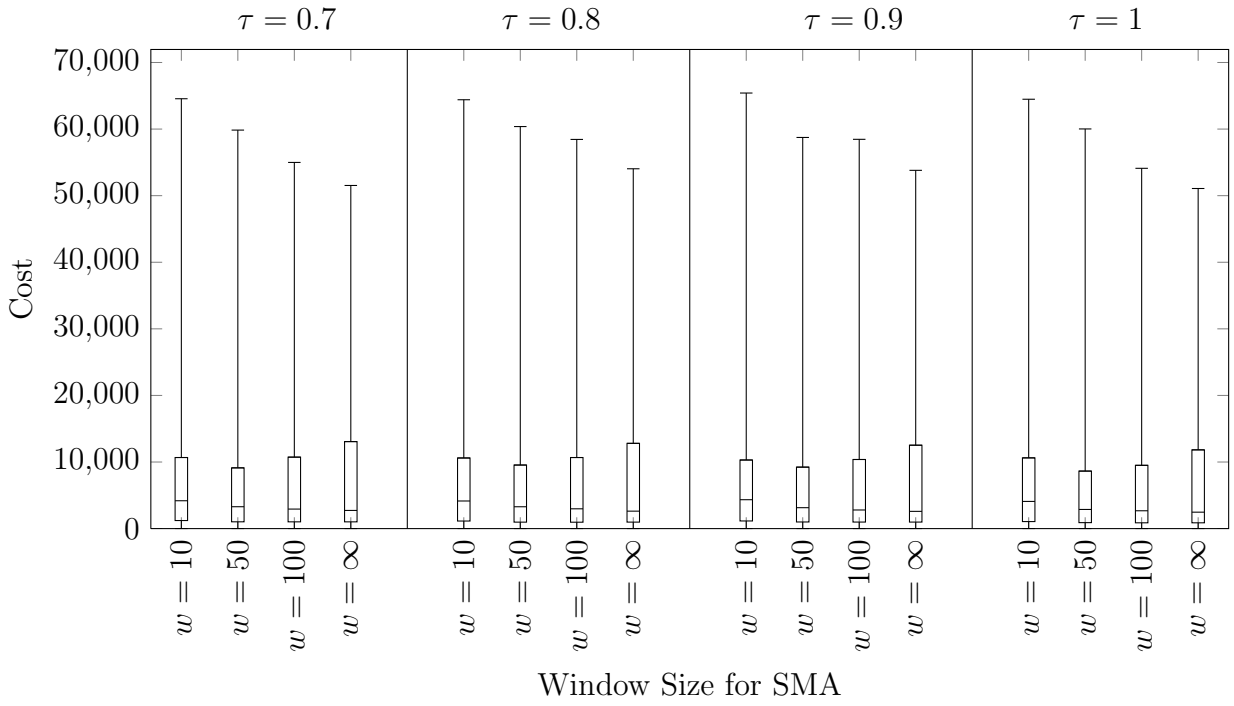

Figure A.5: State Discovery Costs using SMA and Aging for Elfinder

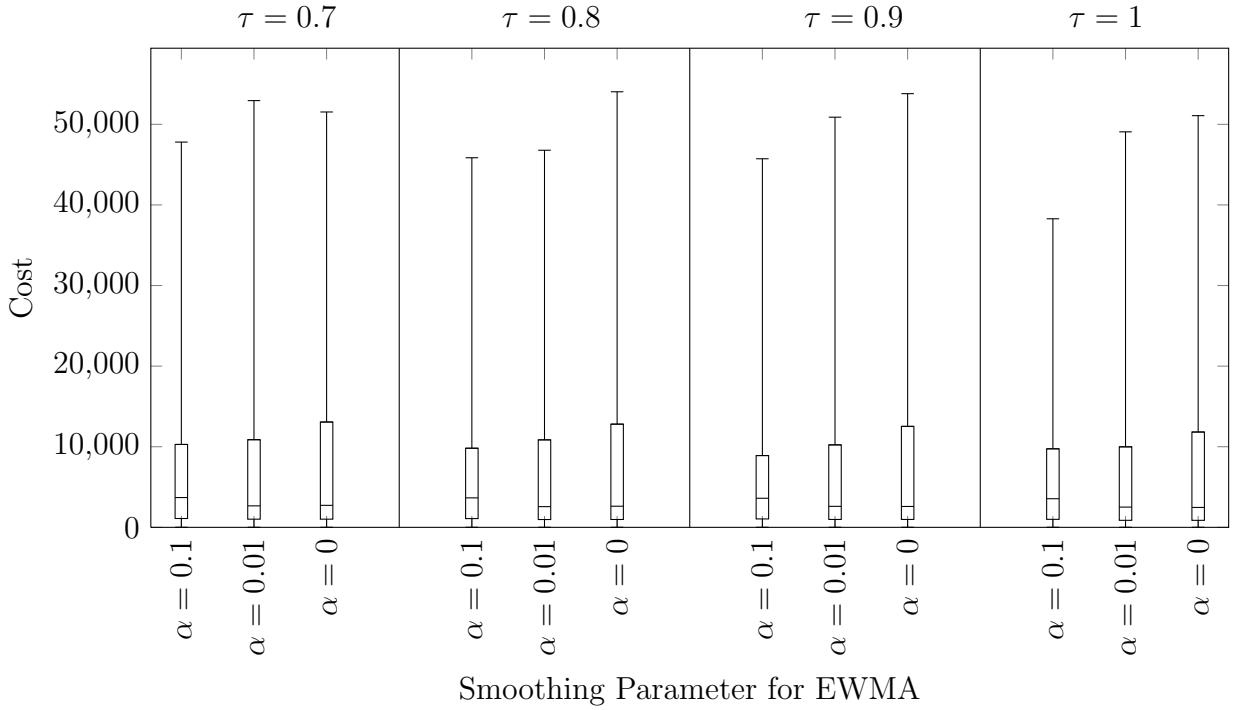

Figure A.6: State Discovery Costs using EWMA and Aging for Elfinder

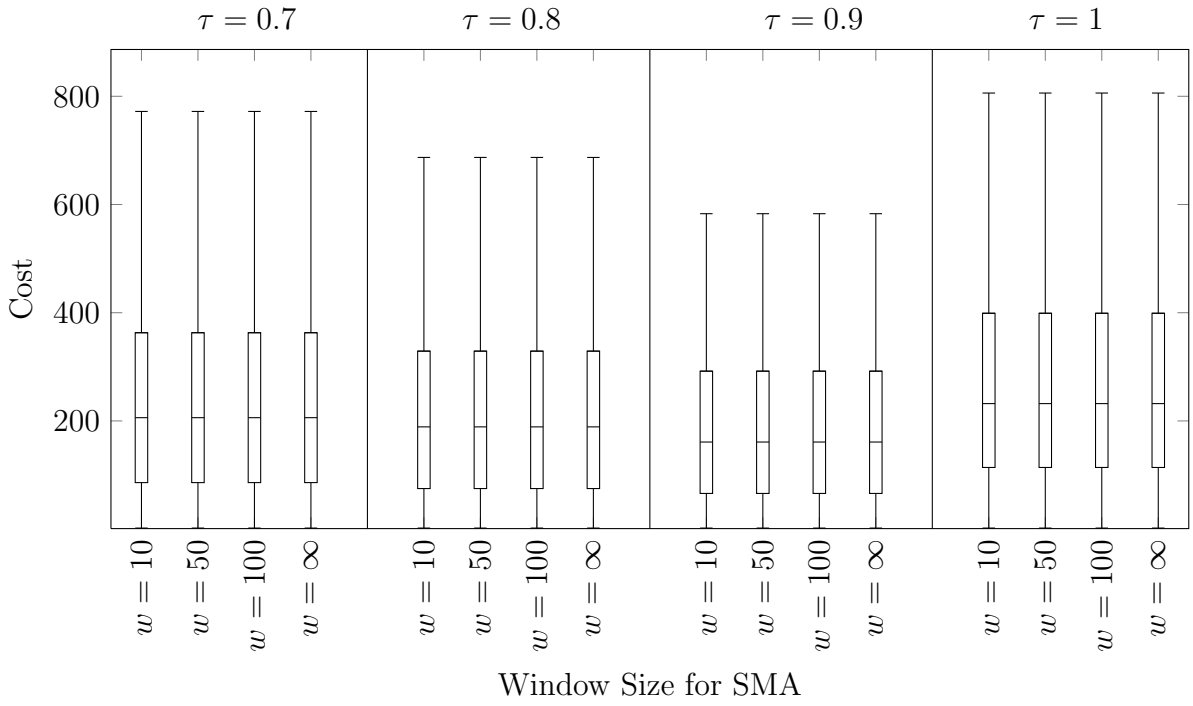

Figure A.7: State Discovery Costs using SMA and Aging for FileTree

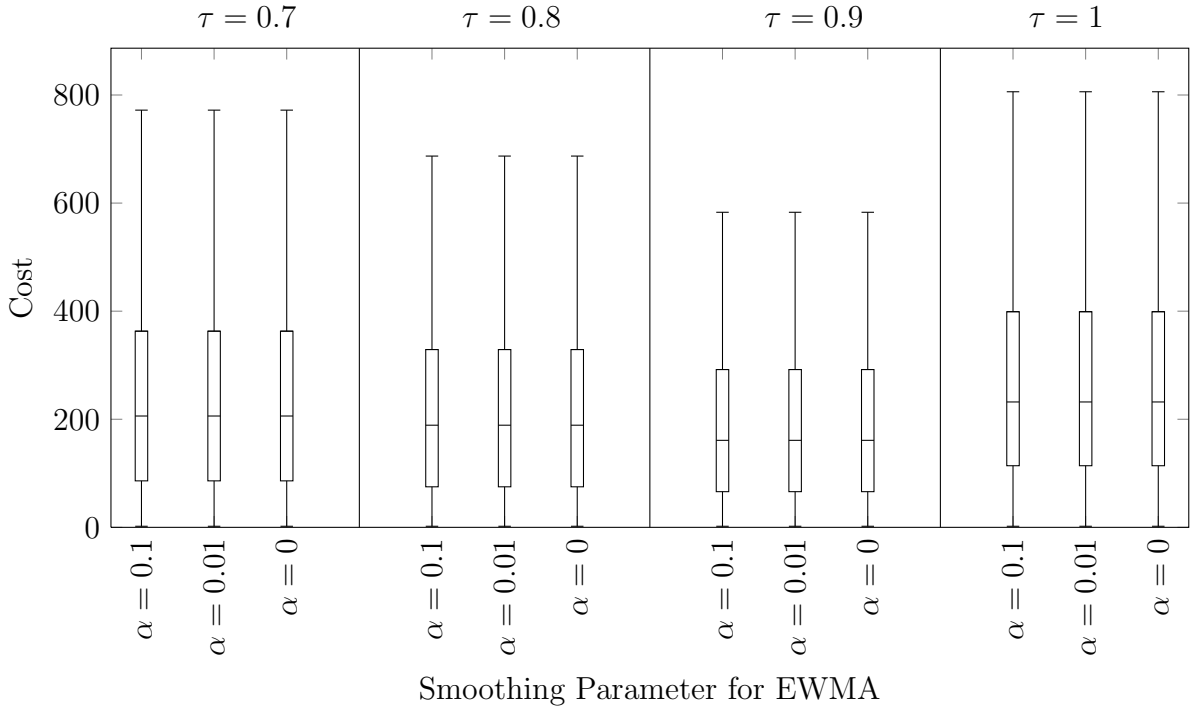

Figure A.8: State Discovery Costs using EWMA and Aging for FileTree

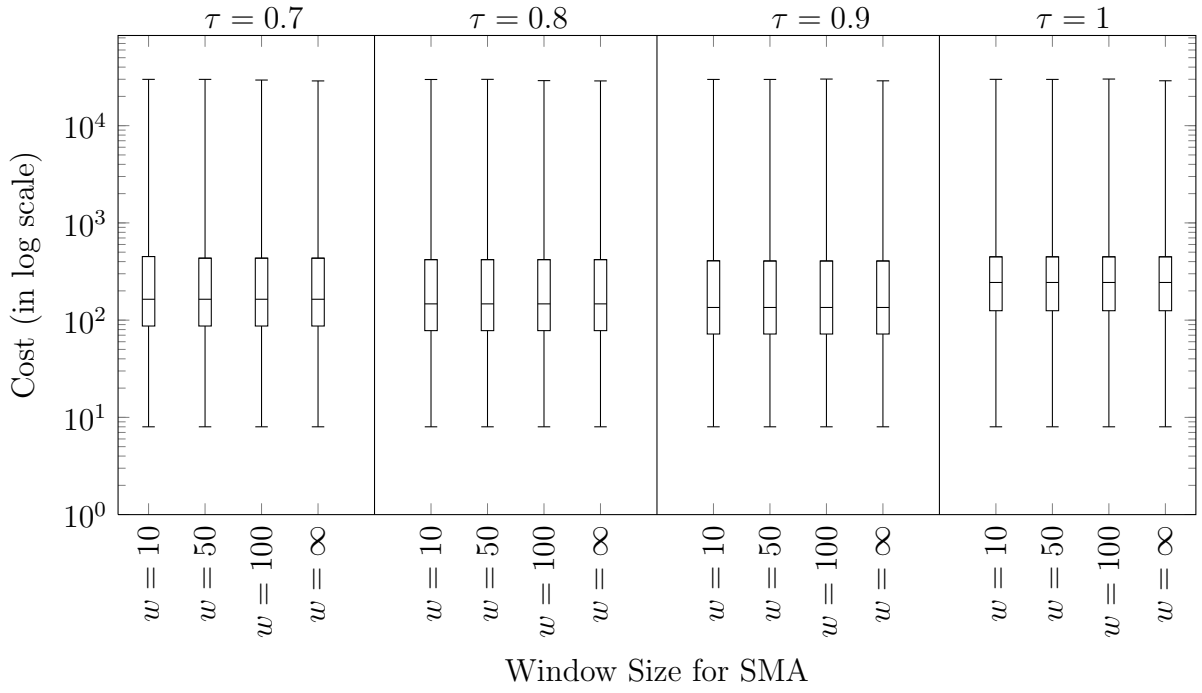

Figure A.9: State Discovery Costs using SMA and Aging for Periodic Table (in log scale)

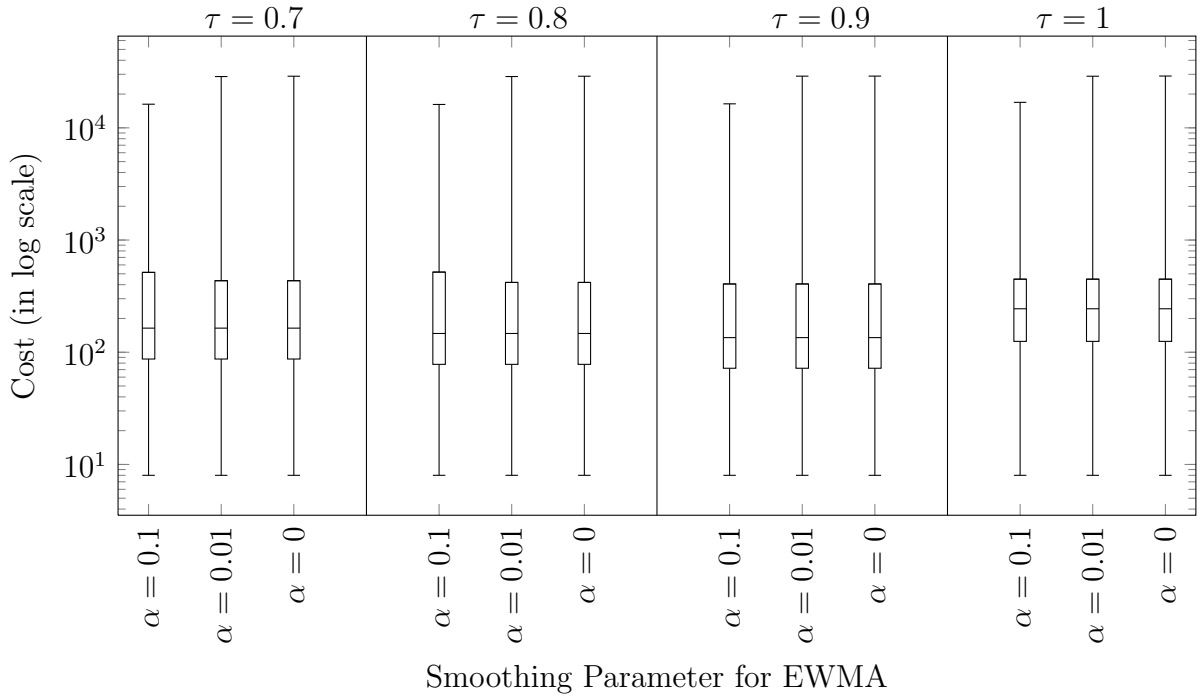

Figure A.10: State Discovery Costs using EWMA and Aging for Periodic Table (in log scale)

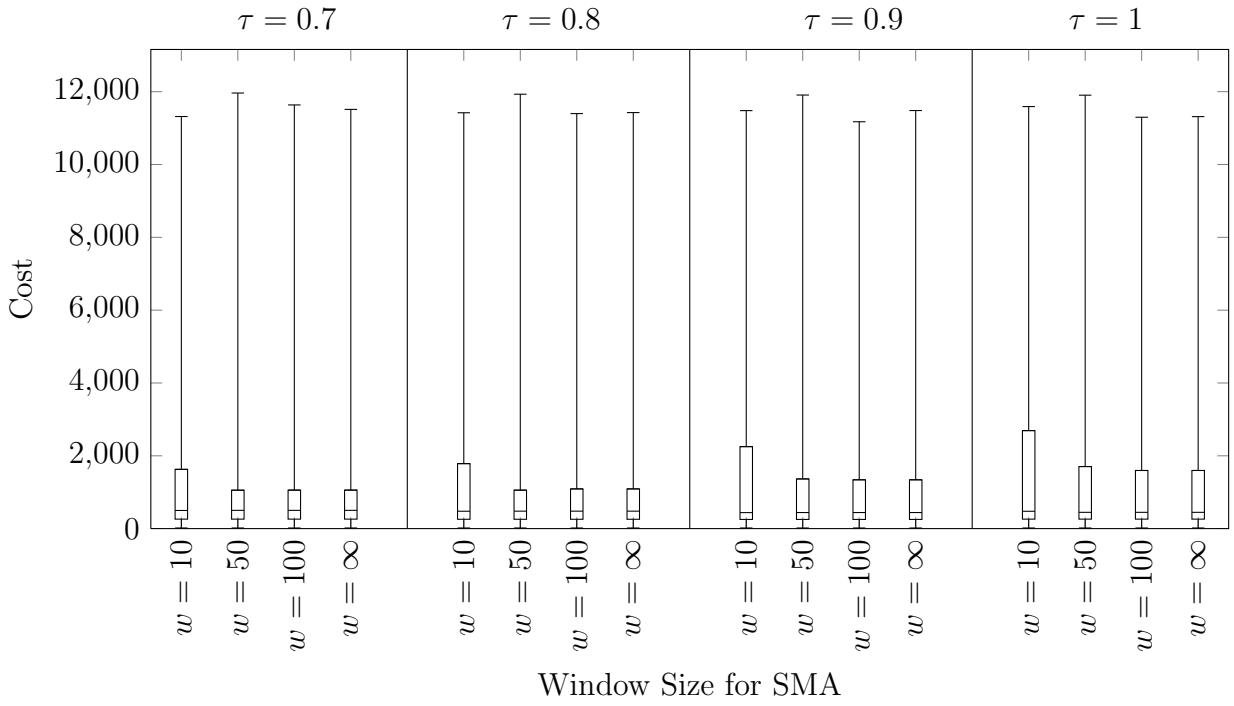

Figure A.11: State Discovery Costs using SMA and Aging for Clipmarks

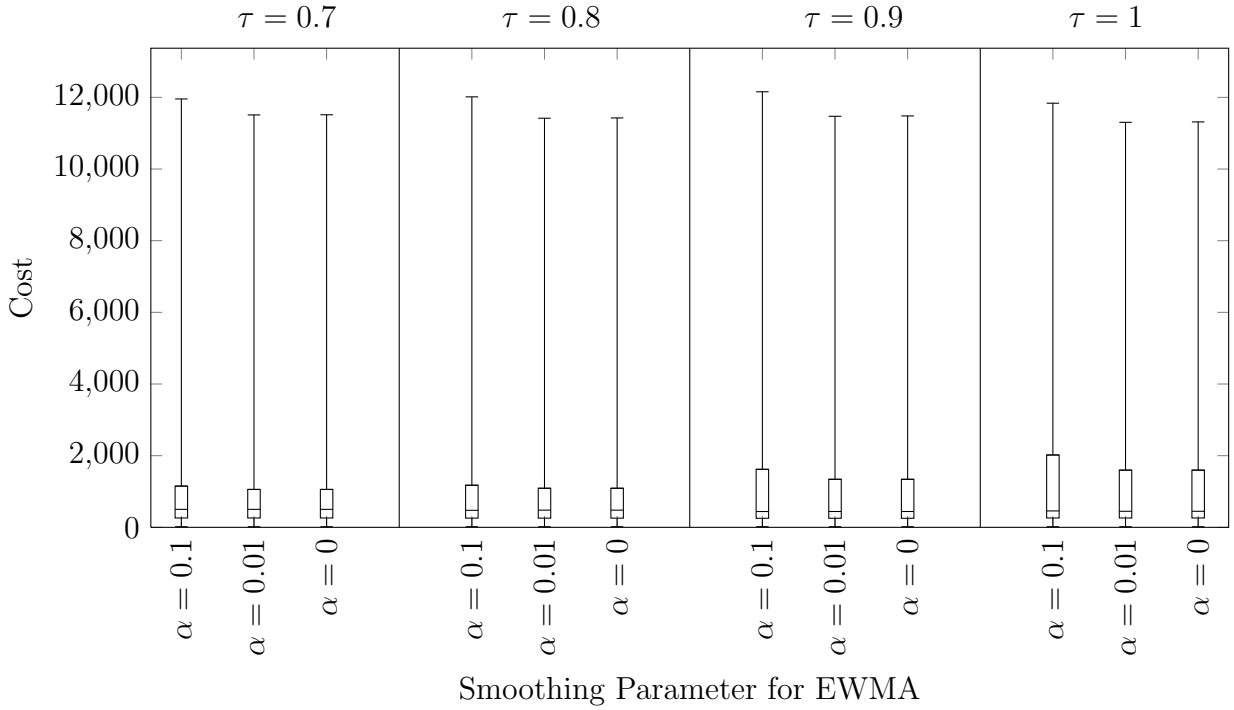

Figure A.12: State Discovery Results using EWMA and Aging for Clipmarks

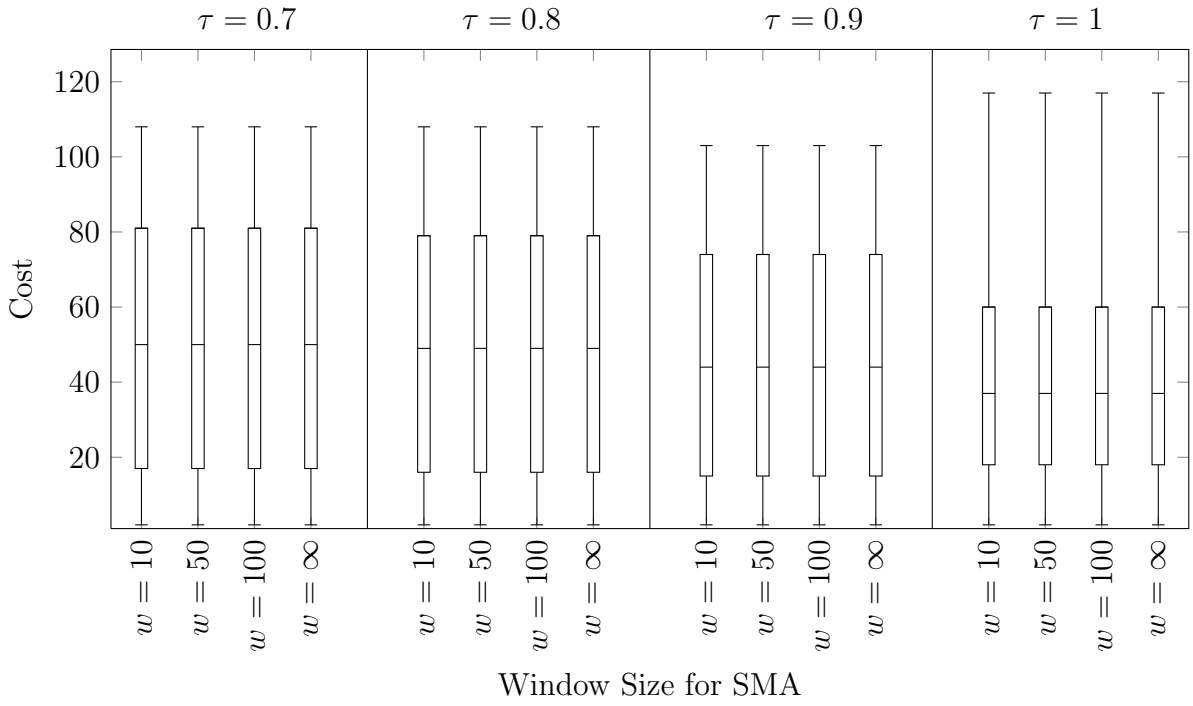

Figure A.13: State Discovery Costs using SMA and Aging for TestRIA

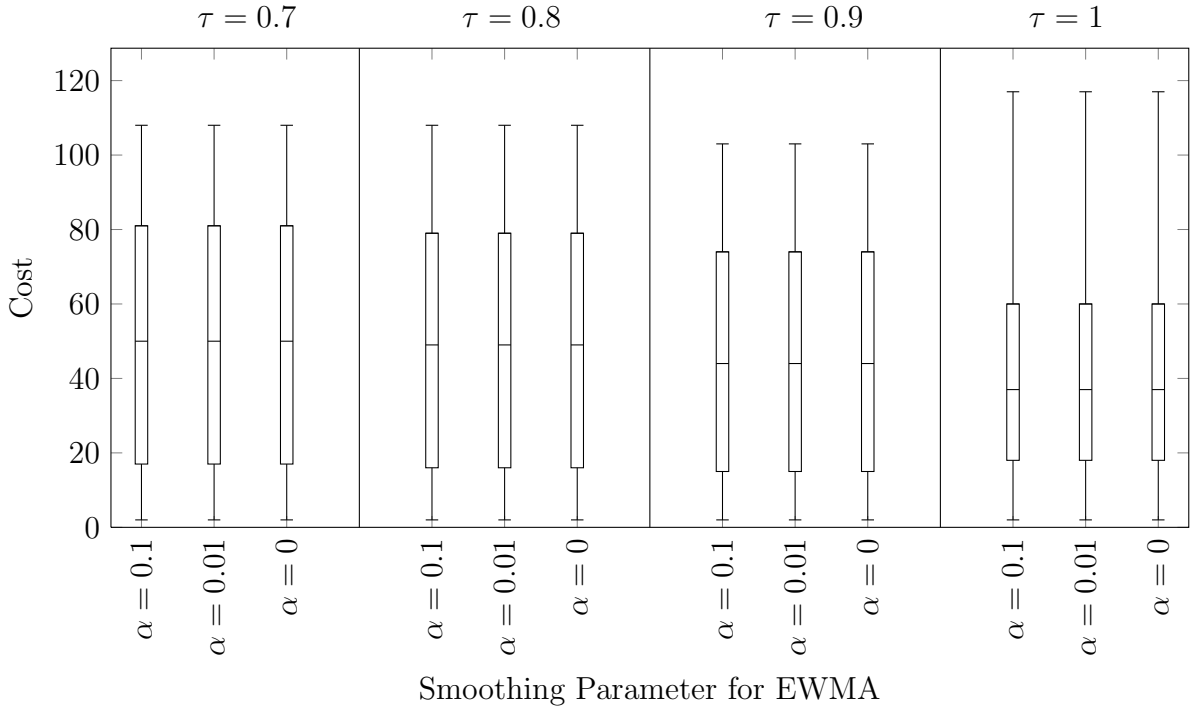

Figure A.14: State Discovery Costs using EWMA and Aging for TestRIA

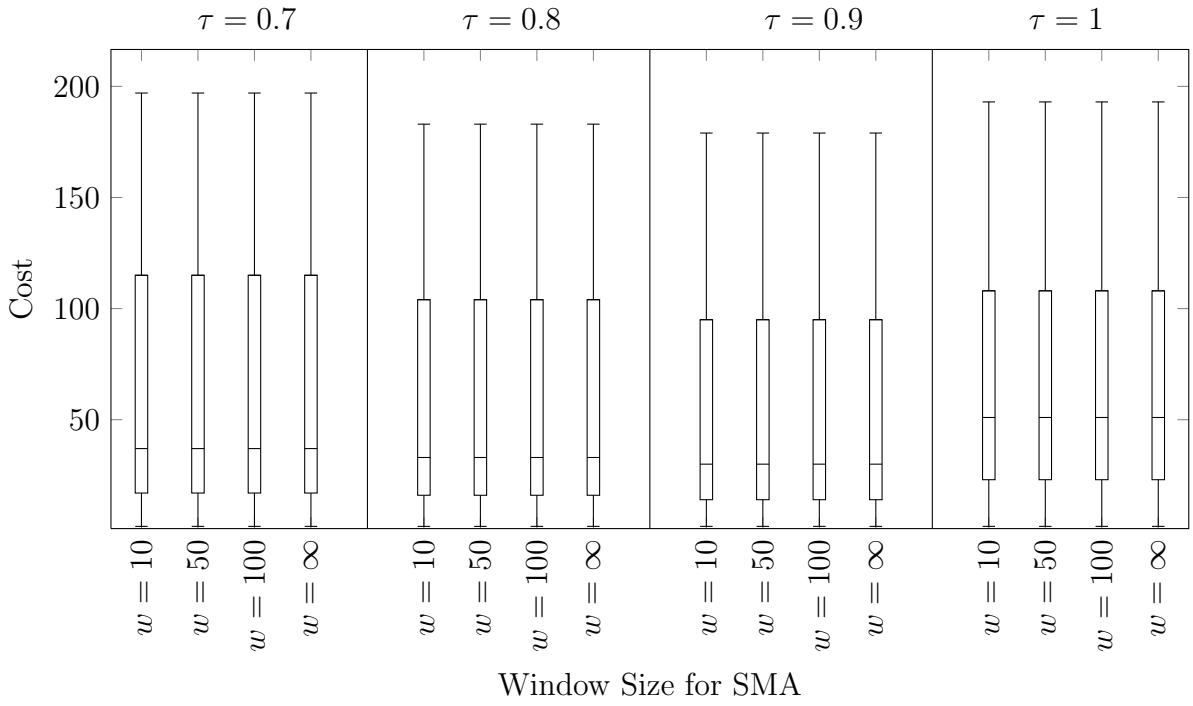

Figure A.15: State Discovery Costs using SMA and Aging for Altoro Mutual

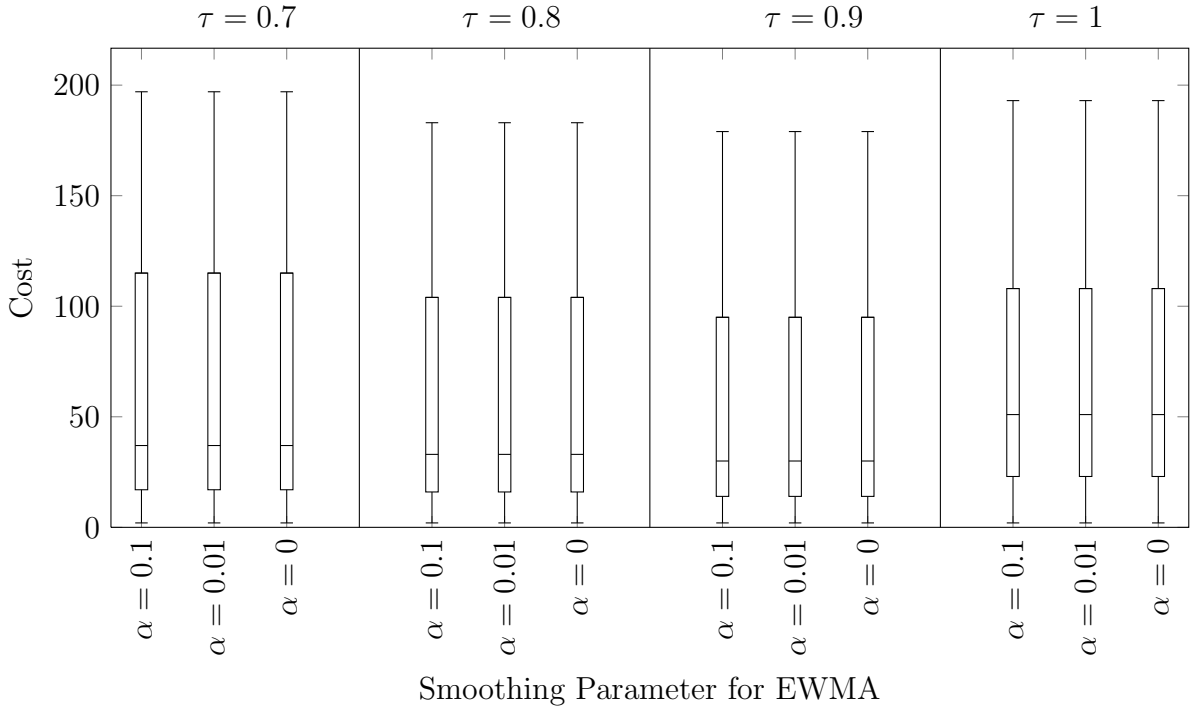

Figure A.16: State Discovery Costs using EWMA and Aging for Altoro Mutual

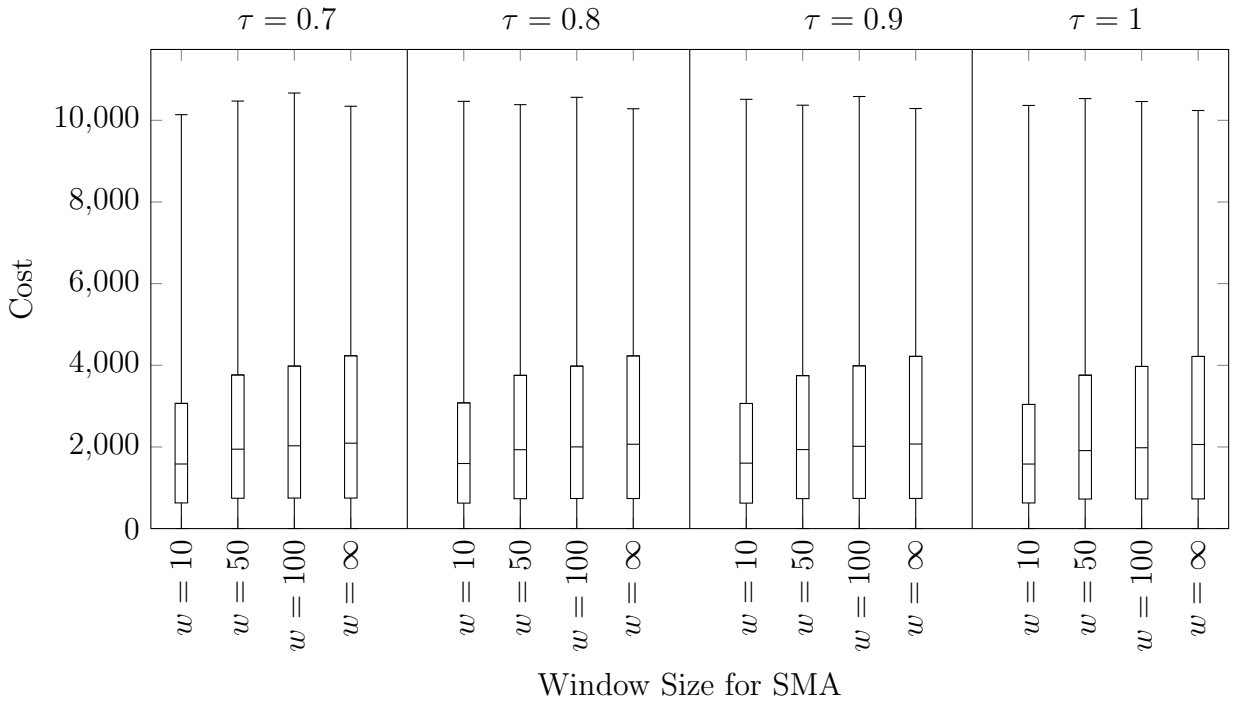

Figure A.17: State Discovery Costs using SMA and Aging for Hypercube10D

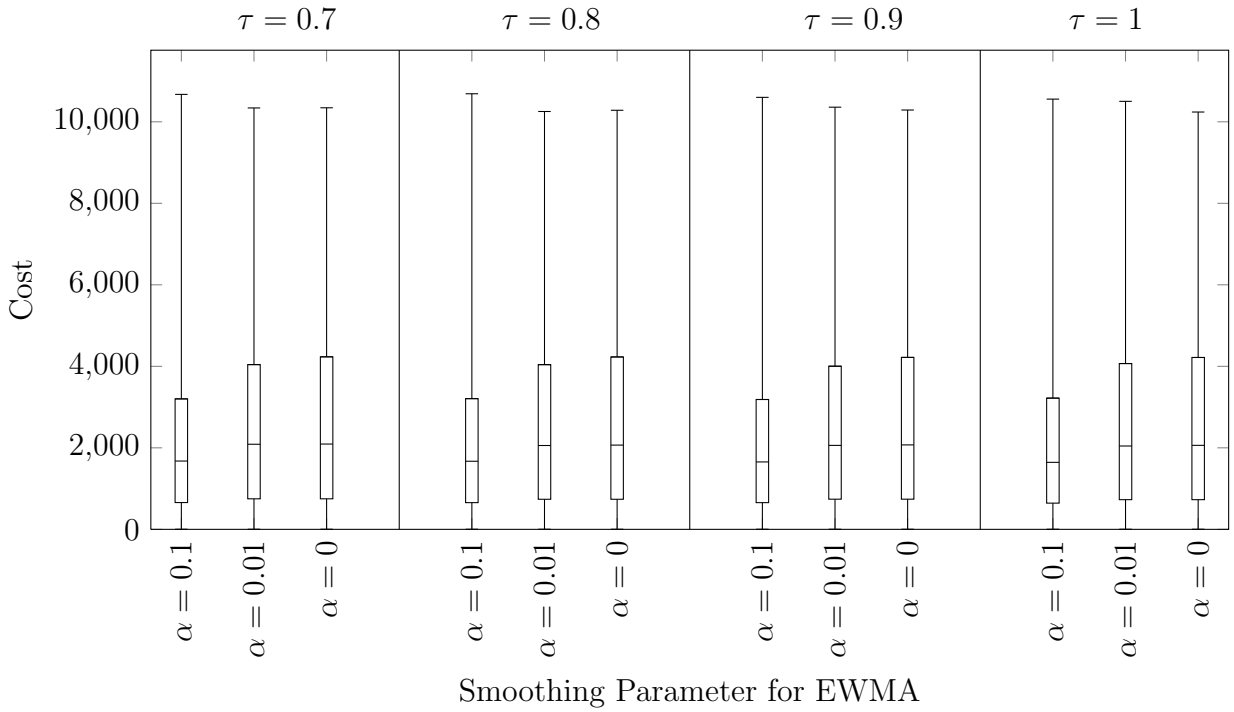

Figure A.18: State Discovery Costs using EWMA and Aging for Hypercube10D

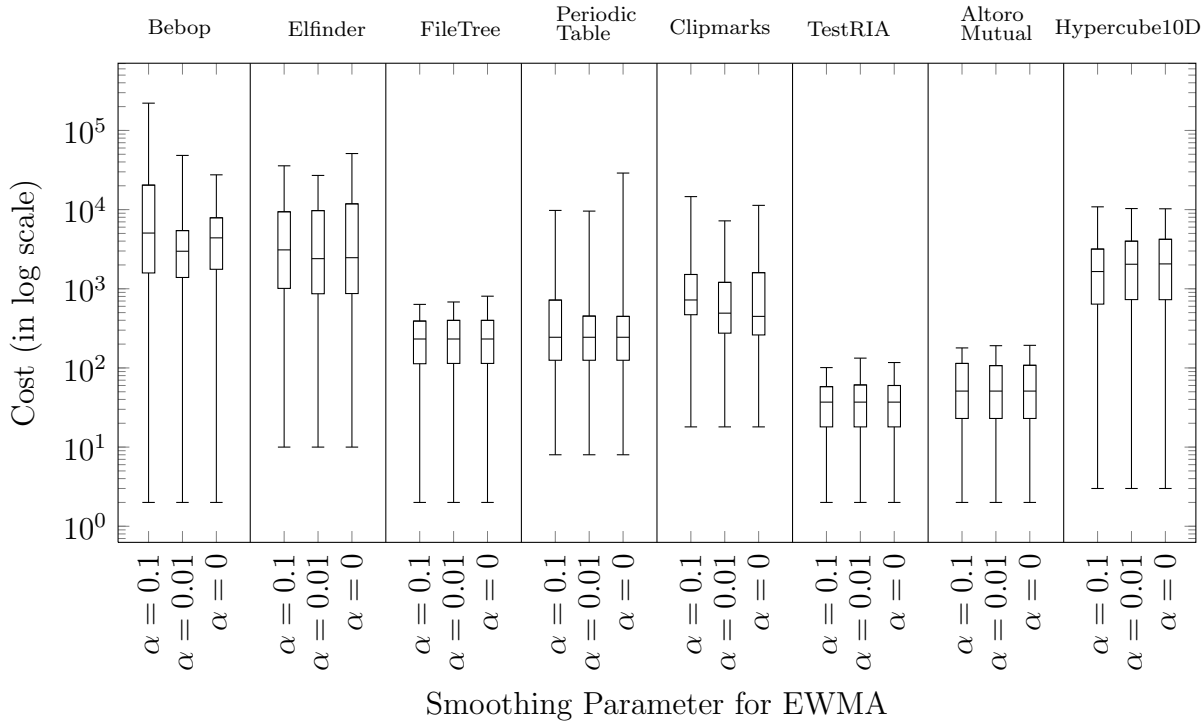

Figure A.19: State Discovery Costs when EWMA is used for both P_{avg} and Event Probabilities (in log scale)

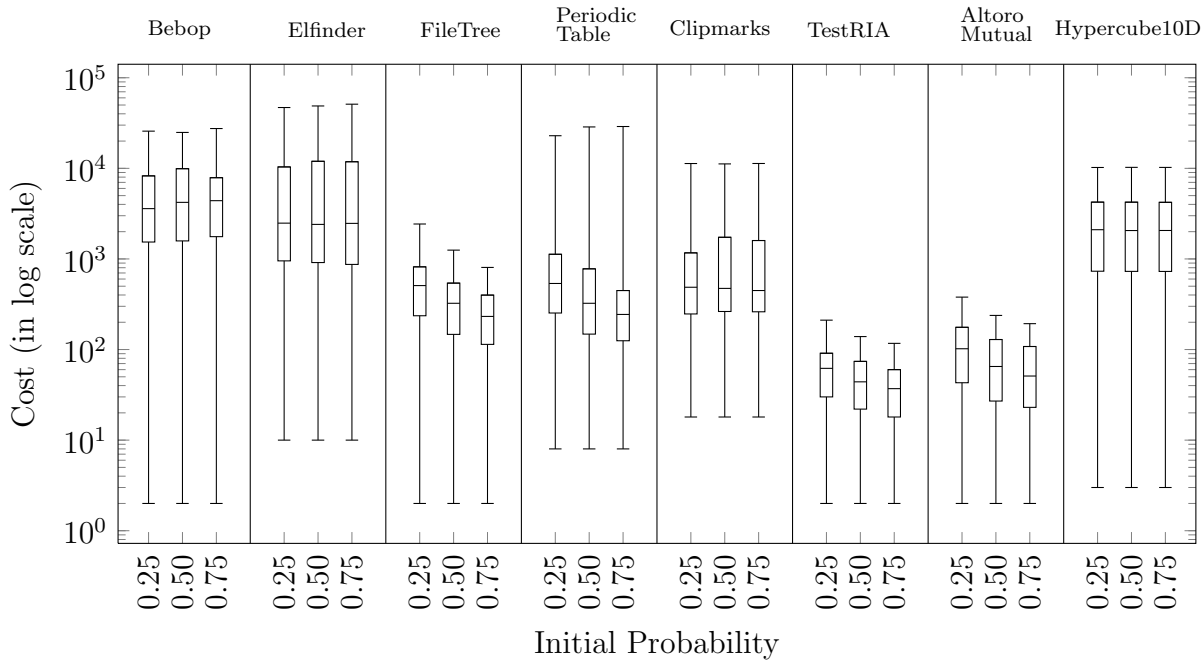

Figure A.20: State Discovery Costs for the Default Strategy with Different Initial Probabilities (in log scale)

Bibliography

[1] Serge Abiteboul, Mihai Preda, and Gregory Cobena. Adaptive on-line page importance computation. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 280–290, New York, NY, USA, 2003. ACM.

[2] Amit Agarwal, Hema Swetha Koppula, Krishna P. Leela, Krishna Prasad Chitrapura, Sachin Garg, Pavan Kumar GM, Chittaranjan Haty, Anirban Roy, and Amit Sasturkar. Url normalization for de-duplication of web pages. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1987–1990, New York, NY, USA, 2009. ACM.

[3] Martin Aigner. Lexicographic matching in boolean algebras. *Journal of Combinatorial Theory*, 14(3):187–194, 1973.

[4] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Reverse engineering finite state machines from rich internet applications. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 69–73, Washington, DC, USA, 2008. IEEE Computer Society.

[5] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 571 –574, sept. 2009.

[6] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Rich internet application testing using execution trace data. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 274–283, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Techniques and tools for rich internet applications testing. In *Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on*, pages 63 –72, sept. 2010.

[8] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 252–261, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[10] Ian Anderson. *Combinatorics of Finite Sets*. Oxford Univ. Press, London, 1987.

[11] Apache. Apache flex. `http://incubator.apache.org/flex/`. [Online].

[12] K.A. Ayoub, H. Aly, and J.M Walsh. Dom based page uniqueness identification. `http://ip.com/patapp/CA2706743A1`, 2010. [Online].

[13] Ricardo Baeza-yates and Carlos Castillo. Balancing volume, quality and freshness in web crawling. In *In Soft Computing Systems - Design, Management and Applications*, pages 565–572. IOS Press, 2002.

[14] Ziv Bar-Yossef, Idit Keidar, and Uri Schonfeld. Do not crawl in the dust: Different urls with similar text. *ACM Trans. Web*, 3(1):3:1–3:31, January 2009.

[15] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 332–345, Washington, DC, USA, 2010. IEEE Computer Society.

[16] Kamara Benjamin. A strategy for efficient crawling of rich internet applications. Master's thesis, EECS - University of Ottawa, 2010. `http://ssrg.eecs.uottawa.ca/docs/Benjamin-Thesis.pdf`.

[17] Kamara Benjamin, Gregor v. Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel Onut. 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 '10, pages 403–409, Washington, DC, USA, 2010. IEEE Computer Society.

[18] Kamara Benjamin, Gregor Von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif Viorel Onut. A strategy for efficient crawling of rich internet applications. In *Proceedings of the 11th international conference on Web engineering*, ICWE'11, pages 74–89, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. 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 '09, pages 81–90, New York, NY, USA, 2009. ACM.

[20] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: a scalable fully distributed web crawler. *Softw. Pract. Exper.*, 34(8):711–726, July 2004.

[21] N.G.D Bruijn, C. Tengbergen, and D. Kruyswijk. On the set of divisors of a number. *Nieuw Arch. Wisk.*, 23:191–194, 1951.

[22] G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Math. Softw.*, 21(4):394–409, December 1995.

[23] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 124–135, New York, NY, USA, 2002. ACM.

[24] Junghoo Cho and Hector Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Trans. Database Syst.*, 28(4):390–426, December 2003.

[25] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 161–172, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[26] Suryakant Choudhary. M-crawler: Crawling rich internet applications using menu meta-model. Master's thesis, EECS - University of Ottawa, 2012. `http://ssrg.site.uottawa.ca/docs/Surya-Thesis.pdf`.

[27] Suryakant Choudhary, Mustafa Emre Dincturk, Gregor V. Bochmann, Guy-Vincent Jourdan, Iosif Viorel Onut, and Paul Ionescu. Solving some modeling challenges when testing rich internet applications for security. *Software Testing, Verification, and Validation, 2012 International Conference on*, 0:850–857, 2012.

[28] Edward G. Coffman, Zhen Liu, and Richard R. Weber. Optimal robot scheduling for web search engines. *Journal of Scheduling*, 1(1):15–29, 1998.

[29] Anirban Dasgupta, Ravi Kumar, and Amit Sasturkar. De-duping urls via rewrite rules. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 186–194, New York, NY, USA, 2008. ACM.

[30] Robert Palmer Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

[31] Stefan Dobrev, Rastislav Krlovi, and Euripides Markou. Online graph exploration with advice. In Guy Even and MagnsM. Halldrsson, editors, *Structural Information and Communication Complexity*, volume 7355 of *Lecture Notes in Computer Science*, pages 267–278. Springer Berlin Heidelberg, 2012.

[32] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: an analysis of black-box web vulnerability scanners. In *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[33] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. Ajax crawl: Making ajax applications searchable. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 78–89, Washington, DC, USA, 2009. IEEE Computer Society.

[34] H. A. Eiselt, Michel Gendreau, and Gilbert Laporte. Arc routing problems, part ii: The rural postman problem. *Operations Research*, 43(3):pp. 399–414, 1995.

[35] M. Erfani and A. Mesbah. Reverse engineering ios mobile applications. In *19th Working Conference on Reverse Engineering, (WCRE'12)*, 2012.

[36] José Exposto, Joaquim Macedo, António Pina, Albano Alves, and José Rufino. Information networking. towards ubiquitous networking and services. chapter Efficient Partitioning Strategies for Distributed Web Crawling, pages 544–553. Springer-Verlag, 2008.

[37] Rudolf Fleischer, Tom Kamphans, Rolf Klein, Elmar Langetepe, and Gerhard Trippen. Competitive online approximation of the optimal search ratio. In *In Proc. 12th Annu. European Sympos. Algorithms, volume 3221 of Lecture Notes Comput. Sci*, pages 335–346. Springer-Verlag, 2004.

[38] Gianni Frey. Indexing ajax web applications. Master's thesis, ETH Zurich, 2007. `http://e-collection.library.ethz.ch/eserv/eth:30111/eth-30111-01.pdf`.

[39] Klaus-Tycho Frster and Roger Wattenhofer. Directed graph exploration. In Roberto Baldoni, Paola Flocchini, and Ravindran Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *Lecture Notes in Computer Science*, pages 151–165. Springer Berlin Heidelberg, 2012.

[40] Jesse James Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, 2005. [Online].

[41] Google. Making ajax applications crawlable. `http://code.google.com/web/ajaxcrawling/index.html`, 2009. [Online].

[42] Curtis Greene and Daniel J. Kleitman. Strong versions of sperner's theorem. *Journal of Combinatorial Theory, Ser. A*, 20(1):80–88, 1976.

[43] Jerrold Griggs, Charles E. Killian, and Carla Savage. Venn diagrams and symmetric chain decompositions in the boolean lattice. *Electron. J. Combin., 11:Research Paper*, 2:21, 2004.

[44] IBM. IBM Security AppScan family. `http://www-01.ibm.com/software/awdtools/appscan/`. [Online].

[45] M. Koster. A standard for robot exclusion. `http://www.robotstxt.org/orig.html`, 1994. [Online].

[46] J Prasanna Kumar and P Govindarajulu. Duplicate and near duplicate documents detection: A review. *European Journal of Scientific Research*, 32:514–527, 2009.

[47] Boon Thau Loo, Loo Owen, and Cooper Sailesh Krishnamurthy. Distributed web crawling over dhts. Technical report, UC Berkeley, 2004.

[48] Jianguo Lu, Yan Wang, Jie Liang, J. Chen, and Jiming Liu. An approach to deep web crawling by sampling. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT '08. IEEE/WIC/ACM International Conference on*, volume 1, pages 718 –724, 2008.

[49] Alessandro Marchetto and Paolo Tonella. Search-based testing of ajax web applications. In *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering*, SSBSE '09, pages 3–12, Washington, DC, USA, 2009. IEEE Computer Society.

[50] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 121–130, Washington, DC, USA, 2008. IEEE Computer Society.

[51] Reto Matter. Ajax crawl: Making ajax applications searchable. Master's thesis, ETH Zurich, 2008. `http://e-collection.library.ethz.ch/eserv/eth:30709/eth-30709-01.pdf`.

[52] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: new results on old and new algorithms. In *Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II*, ICALP'11, pages 478–489, Berlin, Heidelberg, 2011. Springer-Verlag.

[53] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ICWE '08, pages 122–134, Washington, DC, USA, 2008. IEEE Computer Society.

[54] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 210 –220, may 2009.

[55] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *TWEB*, 6(1):3, 2012.

[56] Microsoft. Silverlight. `http://www.microsoft.com/silverlight/`. [Online].

[57] Marc Najork and Janet L. Wiener. Breadth-first crawling yields high-quality pages. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 114–118, New York, NY, USA, 2001. ACM.

[58] Alexandros Ntoulas, Petros Zerfos, and Junghoo Cho. Downloading textual hidden web content through keyword queries. In *Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '05, pages 100–109, New York, NY, USA, 2005. ACM.

[59] Christopher Olston and Marc Najork. Web crawling. *Found. Trends Inf. Retr.*, 4(3):175–246, March 2010.

[60] I.V. Onut, K.A. Ayoub, P. Ionescu, G.v. Bochmann, G.V. Jourdan, M.E Dincturk, and S.M. Mirtaheri. Representation of an element in a page via an identifier. [Patent].

[61] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1998. Standford University, Technical Report.

[62] Zhaomeng Peng, Nengqiang He, Chunxiao Jiang, Zhihua Li, Lei Xu, Yipeng Li, and Yong Ren. Graph-based ajax crawl: Mining data from rich internet applications. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, volume 3, pages 590 –594, march 2012.

[63] K.F. Riley, M.P. Hobson, and S.J. Bence. *Mathematical methods for physics and engineering*. Cambridge University Press, 3rd edition, 2006.

[64] Danny Roest, Ali Mesbah, and Arie van Deursen. Regression testing ajax applications: Coping with dynamism. In *ICST*, pages 127–136. IEEE Computer Society, 2010.

[65] Harold Thimbleby. The directed chinese postman problem. *Software — Practice & Experience*, 33(11):1081–1096, 2003.

[66] World Wide Web Consortium (W3C). Document object model (dom). `http://www.w3.org/DOM/`, 2005. [Online].

[67] Ping Wu, Ji-Rong Wen, Huan Liu, and Wei-Ying Ma. Query selection techniques for efficient crawling of structured web sources. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 47–, Washington, DC, USA, 2006. IEEE Computer Society.

[68] Sandy L. Zabell. The rule of succession. *Erkenntnis*, 31:283–321, 1989.