ABSTRACT

There is a great amount of information on the web that cannot be accessed by conventional crawler engines. This portion of the web is usually called hidden web data. To be able to deal with this problem, it is necessary to solve two tasks: crawling the client-side and crawling the server-side hidden web. In this paper we present an architecture and a set of related techniques for accessing the information placed in the client-side hidden web, dealing with aspects such as JavaScript technology, non-standard session maintenance mechanisms, client redirections, pop-up menus, etc. Our approach leverages current browser APIs and implements novel crawling models and algorithms.

KEYWORDS

Web-Crawler, Hidden Web, Client Side.

1. INTRODUCTION

The part of web documents that is dynamically generated is usually called “Hidden Web” or “Deep Web” [Bergman01]. Crawling the “hidden web” presents two challenges: crawling the client-side and crawling the server-side hidden web. Client-side hidden web techniques are concerned about accessing of content dynamically generated in the client web browser, while server-side techniques are focused in accessing of valuable content hidden behind web search forms [Raghavan01] [Ipeirotis02]. This paper proposes novel techniques and algorithms for dealing with the first of these problems.

Developers use client technologies like scripting languages, session maintenance mechanisms, complex redirections, etc., to add interactivity to web pages as well as for improving site navigation. This presents some problems with conventional crawlers (see [CSHiddenW04] for more detail):
- Successfully dealing with scripting languages requires that HTTP clients implement all the mechanisms that make possible to a browser to render a page and to generate new navigations.
- Many websites use session maintenance mechanisms based on client resources (cookies, scripting code) to add session parameters to the URLs before sending them to the server. This provokes problems when distributing the crawling process and for later accessing of the documents.
- Many websites use complex redirections that are not managed by conventional crawlers.
- Applets and flash are other client technology that conventional crawlers are not able to deal with.

The aforementioned problems are accentuated by issues such as frames, dynamic HTML or HTTPS. We can say that it is very difficult to consider all the factors that make a Website visible and navigable.

1.1 Our approach

This paper presents architecture and a set of related techniques to solve the problems involved in crawling the client-side hidden web. The main features of our approach are the following:
- To solve the problem of session maintenance, our system uses the concept of route to a document, which can be seen as a generalization of the concept of URL (section 3.1).

* This research was partially supported by the Spanish Ministry of Science and Technology under project TIC2001-0547. Alberto Pan’s work was partially supported by the “Ramón y Cajal” programme of the Spanish Ministry of Science and Technology.
To deal with executing scripting code, managing redirections, etc., our crawling processes are not based on http clients, but on automated “mini web browsers” (section 3.2).

To deal with pop-up menus and other dynamic elements that can generate new anchors in the actual page, it is necessary to implement special algorithms to manage the process of generating new “routes to crawl” from a page (section 3.4).

Section 3 describes the architecture and basic functioning of the system. See the extended version of this paper [CSHiddenW04] for more detailed information.

2. INTRODUCTION TO NSEQL

NSEQL [Pan02] is a language to declaratively define sequences of events on the interface provided by a web browser. It allows to easily expressing “macros” representing a sequence of user events over a browser.

NSEQL works “at browser layer” instead of “at HTTP layer”. This lets us forget about problems such as successfully executing JavaScript or dealing with client redirections and session identifiers.

3. THE CRAWLING ENGINE

As well as in conventional crawling engines, the basic idea consists in maintaining a shared list of routes (pointers to documents), which will be accessed by a certain number of concurrent crawling processes, which may be distributed into several machines. The list is initialized and then, each crawling process picks a route from the list, downloads its associated document and analyzes it for obtaining new routes to be added to the master list. The process ends when there are no routes left or when a specified depth level is reached.

3.1 Dealing with sessions: Routes

In conventional crawlers, routes are just URLs. In our system, a route is composed of three elements: A URL pointing to a document (it may be a NSEQL program); a session object containing all the required information (cookies, etc.) for restoring the execution environment that the crawling process had in the moment of adding the route; and a NSEQL program representing the navigation sequence followed to reach the document. The second and third elements are automatically computed by the system for each route. The second element allows a crawling process to access a URL added by other crawling process. The third element is used to access the route when the session originally used to crawl the document has expired or to allow later accessing of them.

3.2 Mini-browsers as crawling processes

Conventional engines use http clients to implement crawling processes. Instead, in our system they are based on automated “mini web browsers”, built using standard browser APIs (current implementation uses MSIE WebBrowser Control [MSIE]). These “mini-browsers” understand NSEQL. This allows our system for:

- Accessing the content dynamically generated through scripting languages.
- Evaluating the scripting code associated with anchors and forms, for obtaining their real URLs.
- Dealing with client redirections, waiting until all the actual page navigation events have finished.
- Providing an execution environment for technologies such as Java applets and Flash. Although the mini-browsers cannot access the content of these “compiled” components, they can deal with the common situation where these components redirect the browser to a conventional page after showing some graphical animation.

3.3 System architecture and basic functioning

Figure 1 shows the architecture of the system. When the crawler engine starts, it reads its configuration parameters from the Configuration Manager module (it includes a list of initial URLs and/or NSEQL
programs, the depth for each initial route, download handlers, content filters, valid DNS domains, etc.).

The following step consists in starting the URL Manager Component with the list of initial sites to crawl, as well as the pool of crawling processes (locally or remotely to the server). The URL Manager is responsible of maintaining the master list of routes to be accessed, and all the crawlers share it. Once the crawling processes have been started, each one picks a route from the URL Manager. Then the crawling process loads the session object associated to the route and downloads the associated document (it uses the Download Manager Component to choose the right handler for the document). If the session has expired, the crawling process will use the NSEQL program instead.

The Content Manager Component is used to analyze the content of each downloaded document. It specifies a chain of filters to decide if the document can be considered relevant and, therefore, if it should be stored. For instance, the system includes filters which allow checking if the document verifies a keyword-based boolean query with a minimum relevance.

Finally, the system tries to obtain new routes from analyzed documents and adds them to the master list. In a context where scripting languages can dynamically generate and remove anchors and forms, this involves some complexities (see section 3.4). The system also includes a chain of filters to decide whether the new routes must be added to the master list or not.

The architecture has components for indexing and searching the crawled contents, using state of the art algorithms. When the user makes a search against the index and the list of answers contains some results which cannot be accessed using its URL due to session issues, the anchors associated to those results in the list will invoke the ActiveX for automatic navigation Component. It receives as a parameter a NSEQL program, downloads itself into the user browser and makes it execute the given navigation sequence.

### 3.4 Algorithm for generating new routes

This section describes the algorithm used to generate new routes to be crawled given a HTML page. This algorithm deals with the difficulties associated to anchors and forms controlled by scripting languages.

In general, to get the new routes to be crawled from a given HTML document, it is necessary to analyze the page looking for anchors and client maps. A new route will be added for each anchor and for each map area. The anchors and maps, which are not controlled by scripting code, can be dealt with as in conventional crawlers. Nevertheless, if the HTML page contains client-side scripting technology, the situation is more complicated. The main idea of the algorithm consists on generating click events over the anchors controlled by scripting languages in order to obtain valid URLs (NOTE: we will focus our discussion on the case of anchors. The treatment of maps would be analogous). There are several difficulties:

- Some anchors may appear or disappear from the page depending on the scripting code executed.
- The script code associated to anchors must be evaluated in order to obtain valid URLs.
- One anchor can generate several navigations.

Now we proceed to describe the algorithm. Remind our crawling process is a “mini-browser” able to execute NSEQL programs. The browser can be in two states: in the navigation state the browser functions normally, and when it executes a click event on an anchor or submits a form, it performs the navigation and downloads the resulting page; in turn, in the simulation state the browser only captures the navigation events generated by the click or submit events, but it does not download the resource.

1. Let $P$ be an HTML page that has been downloaded by the browser (navigation state).
2. The browser executes the scripting sections which are not associated to conditional events.
3. Let \( A_p \) be all the anchors of the page with the scripting code already interpreted.

4. For each \( a_i \in A_p \), remove \( a_i \)
   a) If the \( href \) attribute from \( a_i \) does not contain associated scripting code and it has not an \( onclick \) attribute (or other attributes such as \( onmouseover \), if defined), the anchor \( a_i \) is added to the master list of URLs.
   b) Otherwise, the browser generates a \( click \) event on the anchor -other event, if defined- (simulation state):
   a. There exist some anchors that, when clicked, can generate undesirable actions (e.g. a call to “javascript:close”). The approach followed to avoid this is to capture and ignore these events.
   b. The crawler captures all the new navigation events that happen after the click. Each navigation event produces an URL. Let \( A_p \) be the set of all the new URLs.
   c. \( A_p = A_p \cup A_p \).
   d. Once the execution of the events associated to a click over an anchor has finished, the crawler analyzes the page again looking for new anchors that could have been generated by the click event (e.g., new options corresponding to pop-up menus), \( A_{np} \). New anchors are added to \( A_p \). \( A_p = A_{np} \cup A_p \).

5. The browser changes to navigation state, and the crawler is ready to process a new URL.

If the processed page has several frames, then the system will process each frame in the same way.

Note that the system processes the anchors in a page following a bottom-up approach, so new anchors are added on the list before the existing ones. In this way, new anchors will be processed before some other click can remove them from the page. Also note that the added anchors will have to agree with the filters for adding URLs mentioned in section 3.3.

4. RELATED WORK AND CONCLUSIONS

A well-known approach for discovering and indexing relevant information is to “crawl” a given information space looking for information verifying certain requirements. Nevertheless, today’s web “crawlers” or “spiders” [Brin98] do not deal with the hidden web. During the last few years, there have been some pioneer research efforts dealing with the complexities of accessing the server-side hidden web [Raghavan01] [Ipeirotis02]. Some crawling systems [WebCopier] have included JavaScript interpreters [Rhino] in the HTTP clients they use to provide some limited support for dealing with JavaScript. Nevertheless, our system offer several advantages:

- It is able to correctly execute any scripting code in the same manner as a conventional web browser.
- It is able to deal with session maintenance mechanisms for both crawling and later accessing of documents (using the ActiveX component).
- It is able to deal with anchors and forms dynamically generated in response to user events.
- It is able to deal with redirections (including those generated by Java applets and Flash programs).

Finally, we want to remark that the system presented in this paper has already been successfully used in several real-world applications in fields such as corporate search and technology watch.

REFERENCES