Crawling the Web

Gautam Pant[1], Padmini Srinivasan[1,2], and Filippo Menczer[3]

[1] Department of Management Sciences
[2] School of Library and Information Science
   The University of Iowa, Iowa City IA 52242, USA
   {gautam-pant,padmini-srinivasan}@uiowa.edu
[3] School of Informatics
   Indiana University, Bloomington, IN 47408, USA
   fil@indiana.edu

Summary. The large size and the dynamic nature of the Web make it necessary to continually maintain Web based information retrieval systems. Crawlers facilitate this process by following hyperlinks in Web pages to automatically download new and updated Web pages. While some systems rely on crawlers that exhaustively crawl the Web, others incorporate "focus" within their crawlers to harvest application- or topic-specific collections. In this chapter we discuss the basic issues related to developing an infrastructure for crawlers. This is followed by a review of several topical crawling algorithms, and evaluation metrics that may be used to judge their performance. Given that many innovative applications of Web crawling are still being invented, we briefly discuss some that have already been developed.

1 Introduction

Web *crawlers* are programs that exploit the graph structure of the Web to move from page to page. In their infancy such programs were also called wanderers, robots, spiders, fish, and worms, words that are quite evocative of Web imagery. It may be observed that the noun "crawler" is not indicative of the speed of these programs, as they can be considerably fast. In our own experience, we have been able to crawl up to tens of thousands of pages within a few minutes while consuming a small fraction of the available bandwidth.[4]

From the beginning, a key motivation for designing Web crawlers has been to retrieve Web pages and add them or their representations to a local repository. Such a repository may then serve particular application needs such as those of a Web search engine. In its simplest form a crawler starts from a *seed* page and then uses the external links within it to attend to other pages. The process repeats with the new pages

---

[4] We used a Pentium 4 workstation with an Internet2 connection.

offering more external links to follow, until a sufficient number of pages are identified or some higher-level objective is reached. Behind this simple description lies a host of issues related to network connections, spider traps, canonicalizing URLs, parsing HTML pages, and the ethics of dealing with remote Web servers. In fact, a current generation Web crawler can be one of the most sophisticated yet fragile parts [5] of the application in which it is embedded.

Were the Web a static collection of pages we would have little long-term use for crawling. Once all the pages had been fetched to a repository (like a search engine's database), there would be no further need for crawling. However, the Web is a dynamic entity with subspaces evolving at differing and often rapid rates. Hence there is a continual need for crawlers to help applications stay current as new pages are added and old ones are deleted, moved or modified.

General-purpose search engines serving as entry points to Web pages strive for coverage that is as broad as possible. They use Web crawlers to maintain their index databases [3], amortizing the cost of crawling and indexing over the millions of queries received by them. These crawlers are blind and exhaustive in their approach, with comprehensiveness as their major goal. In contrast, crawlers can be selective about the pages they fetch and are then referred to as *preferential* or heuristic-based crawlers [10, 6]. These may be used for building focused repositories, automating resource discovery, and facilitating software agents. There is a vast literature on preferential crawling applications including [15, 9, 31, 20, 26, 3]. Preferential crawlers built to retrieve pages within a certain topic are called *topical* or *focused* crawlers. Synergism between search engines and topical crawlers is certainly possible, with the latter taking on the specialized responsibility of identifying subspaces relevant to particular communities of users. Techniques for preferential crawling that focus on improving the "freshness" of a search engine have also been suggested [3].

Although a significant portion of this chapter is devoted to description of crawlers in general, the overall slant, particularly in the latter sections, is toward topical crawlers. There are several dimensions about topical crawlers that make them an exciting object of study. One key question that has motivated much research is: How is crawler selectivity to be achieved? Rich contextual aspects, such as the goals of the parent application, lexical signals within the Web pages, and also features of the graph built from pages already seen, are all reasonable kinds of evidence to exploit. Additionally, crawlers can and often do differ in their mechanisms for using the evidence available to them.

A second major aspect that is important to consider when studying crawlers, especially topical crawlers, is the nature of the crawl task. Crawl characteristics such as queries and/or keywords provided as input criteria to the crawler, user-profiles, and desired properties of the pages to be fetched (similar pages, popular pages, authoritative pages, etc.) can lead to significant differences in crawler design and implementation. The task could be constrained by parameters like the maximum number of pages to be fetched (long crawls versus short crawls) or the available memory. Hence, a crawling task can be viewed as a constrained multiobjective search problem. However, the wide variety of objective functions, coupled with the lack of appropriate knowledge about

the search space, make the problem a hard one. Furthermore, a crawler may have to deal with optimization issues such as local versus global optima [28].

The last key dimension is regarding crawler evaluation strategies necessary to make comparisons and determine circumstances under which one or the other crawlers work best. Comparisons must be fair and be made with an eye toward drawing out statistically significant differences. Not only does this require a sufficient number of crawl runs but also sound methodologies that consider the temporal nature of crawler outputs. Significant challenges in evaluation include the general unavailability of relevant sets for particular topics or queries. Thus evaluation typically relies on defining measures for estimating page importance.

The first part of this chapter presents a crawling infrastructure and within this describes the basic concepts in Web crawling. Following this, we review a number of crawling algorithms that are suggested in the literature. We then discuss current methods to evaluate and compare performance of different crawlers. Finally, we outline the use of Web crawlers in some applications.

2 Building a Crawling Infrastructure

Figure 1 shows the flow of a basic sequential crawler (in Sect. 2.6 we consider multi-threaded crawlers). The crawler maintains a list of unvisited URLs called the *frontier*. The list is initialized with seed URLs, which may be provided by a user or another program. Each *crawling loop* involves picking the next URL to crawl from the frontier, fetching the page corresponding to the URL through HTTP, parsing the retrieved page to extract the URLs and application-specific information, and finally adding the unvisited URLs to the frontier. Before the URLs are added to the frontier they may be assigned a score that represents the estimated benefit of visiting the page corresponding to the URL. The crawling process may be terminated when a certain number of pages have been crawled. If the crawler is ready to crawl another page and the frontier is empty, the situation signals a deadend for the crawler. The crawler has no new page to fetch, and hence it stops.

Crawling can be viewed as a graph search problem. The Web is seen as a large graph with pages at its nodes and hyperlinks as its edges. A crawler starts at a few of the nodes (seeds) and then follows the edges to reach other nodes. The process of fetching a page and extracting the links within it is analogous to expanding a node in graph search. A topical crawler tries to follow edges that are expected to lead to portions of the graph that are relevant to a topic.

2.1 Frontier

The frontier is the to-do list of a crawler that contains the URLs of unvisited pages. In graph search terminology the frontier is an *open list* of unexpanded (unvisited) nodes. Although it may be necessary to store the frontier on disk for large -scale crawlers, we will represent the frontier as an in-memory data structure for simplicity. Based on the available memory, one can decide the maximum size of the frontier. Because of the
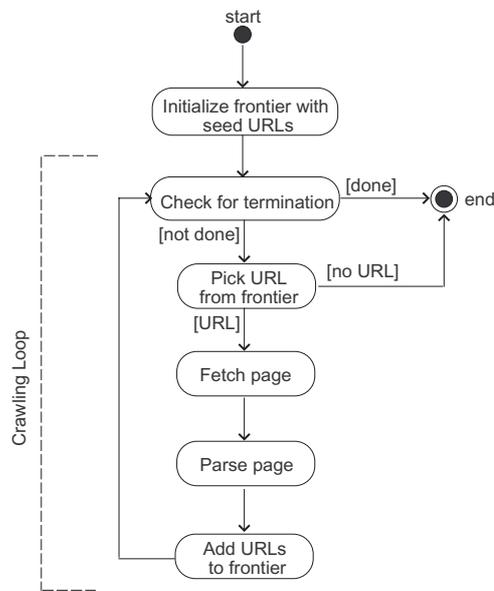
Fig. 1. Flow of a basic sequential crawler

large amount of memory available on PCs today, a frontier size of a 100,000 URLs or more is not exceptional. Given a maximum frontier size we need a mechanism to decide which URLs to ignore when this limit is reached. Note that the frontier can fill rather quickly as pages are crawled. One can expect around 60,000 URLs in the frontier with a crawl of 10,000 pages, assuming an average of about 7 links per page [30].

The frontier may be implemented as a FIFO queue, in which case we have a breadth-first crawler that can be used to blindly crawl the Web. The URL to crawl next comes from the head of the queue, and the new URLs are added to the tail of the queue. Because of the limited size of the frontier, we need to make sure that we do not add duplicate URLs into the frontier. A linear search to find out if a newly extracted URL is already in the frontier is costly. One solution is to allocate some amount of available memory to maintain a separate hash-table (with URL as key) to store each of the frontier URLs for fast lookup. The hash-table must be kept synchronized with the actual frontier. A more time -consuming alternative is to maintain the frontier itself as a hash-table (again with URL as key). This would provide fast lookup for avoiding duplicate URLs. However, each time the crawler needs a URL to crawl, it would need to search and pick the URL with the earliest time stamp (the time when a URL was added to the frontier). If memory is less of an issue than speed, the first solution may be preferred. Once the frontier reaches its maximum size, the breadth-first crawler can add only one unvisited URL from each new page crawled.

If the frontier is implemented as a priority queue we have a preferential crawler, which is also known as a best-first crawler. The priority queue may be a dynamic

array that is always kept sorted by the estimated score of unvisited URLs. At each step, the best URL is picked from the head of the queue. Once the corresponding page is fetched, the URLs are extracted from it and scored based on some heuristic. They are then added to the frontier in such a manner that the order of the priority queue is maintained. We can avoid duplicate URLs in the frontier by keeping a separate hash-table for lookup. Once the frontier's maximum size (MAX) is exceeded, only the best MAX URLs are kept in the frontier.

If the crawler finds the frontier empty when it needs the next URL to crawl,the crawling process comes to a halt. With a large value of MAX and several seed URLs the frontier will rarely reach the empty state.

At times, a crawler may encounter a *spider trap* that leads it to a large number of different URLs that refer to the same page. One way to alleviate this problem is by limiting the number of pages that the crawler accesses from a given domain. The code associated with the frontier can make sure that every consecutive sequence of k (say 100) URLs, picked by the crawler, contains only one URL from a fully qualified host name (e.g., www.cnn.com). As side effects, the crawler is polite by not accessing the same Web site too often [14], and the crawled pages tend to be more diverse.

2.2 History and Page Repository

The crawl history is a time-stamped list of URLs that were fetched by the crawler. In effect, it shows the path of the crawler through the Web, starting from the seed pages. A URL entry is made into the history only after fetching the corresponding page. This history may be used for post-crawl analysis and evaluations. For example, we can associate a value with each page on the crawl path and identify significant events (such as the discovery of an excellent resource). While history may be stored occasionally to the disk, it is also maintained as an in-memory data structure. This provides for a fast lookup to check whether a page has been crawled or not. This check is important to avoid revisiting pages and also to avoid adding the URLs of crawled pages to the limited size frontier. For the same reasons it is important to *canonicalize* the URLs (Sect. 2.4) before adding them to the history.

Once a page is fetched, it may be stored/indexed for the master application (such as a search engine). In its simplest form a *page repository* may store the crawled pages as separate files. In that case, each page must map to a unique file name. One way to do this is to map each page's URL to a compact string using some form of hashing function with low probability of collisions (for uniqueness of file names). The resulting hash value is used as the file name. We use the MD5 one-way hashing function that provides a 128-bit hash code for each URL. Implementations of MD5 and other hashing algorithms are readily available in different programming languages (e.g., refer to Java 2 security framework[5]). The 128-bit hash value is then converted into a 32-character hexadecimal equivalent to get the file name. For example, the content of http://www.uiowa.edu/ is stored into a file named 160766577426e1d01fcb7735091ec584. This way we have fixed-length file

---

[5] http://java.sun.com

names for URLs of arbitrary size. (Of course, if the application needs to cache only a few thousand pages, one may use a simpler hashing mechanism.) The page repository can also be used to check if a URL has been crawled before by converting it to its 32-character file name and checking for the existence of that file in the repository. In some cases this may render unnecessary the use of an in-memory history data structure.

2.3 Fetching

In order to fetch a Web page, we need an HTTP client that sends an HTTP request for a page and reads the response. The client needs to have timeouts to make sure that an unnecessary amount of time is not spent on slow servers or in reading large pages. In fact, we may typically restrict the client to download only the first 10–20KB of the page. The client needs to parse the response headers for status codes and redirections. We may also like to parse and store the last-modified header to determine the age of the document. Error checking and exception handling are important during the page-fetching process since we need to deal with millions of remote servers using the same code. In addition, it may be beneficial to collect statistics on timeouts and status codes for identifying problems or automatically changing timeout values. Modern programming languages such as Java and Perl provide very simple and often multiple programmatic interfaces for fetching pages from the Web. However, one must be careful in using high-level interfaces where it may be harder to find lower-level problems. For example, with Java one may want to use the `java.net.Socket` class to send HTTP requests instead of using the more ready-made `java.net.HttpURLConnection` class.

No discussion about crawling pages from the Web can be complete without talking about the *Robot Exclusion Protocol*. This protocol provides a mechanism for Web server administrators to communicate their file access policies; more specifically to identify files that may not be accessed by a crawler. This is done by keeping a file named `robots.txt` under the root directory of the Web server (such as `http://www.biz.uiowa.edu/robots.txt`). This file provides access policy for different *User-agents* (robots or crawlers). A User-agent value of "*" denotes a default policy for any crawler that does not match other User-agent values in the file. A number of *Disallow* entries may be provided for a User-agent. Any URL that starts with the value of a Disallow field must not be retrieved by a crawler matching the User-agent. When a crawler wants to retrieve a page from a Web server, it must first fetch the appropriate `robots.txt` file and make sure that the URL to be fetched is not disallowed. More details on this exclusion protocol can be found at `http://www.robotstxt.org/wc/norobots.html`. It is efficient to cache the access policies of a number of servers recently visited by the crawler. This would avoid accessing a `robots.txt` file each time you need to fetch a URL. However, one must make sure that cache entries remain sufficiently fresh.

2.4 Parsing

Once a page has been fetched, we need to parse its content to extract information that will feed and possibly guide the future path of the crawler. Parsing may imply simple hyperlink/URL extraction or it may involve the more complex process of tidying up the HTML content in order to analyze the HTML tag tree (Sect. 2.5). Parsing might also involve steps to convert the extracted URL to a canonical form, remove stopwords from the page's content, and stem the remaining words. These components of parsing are described next.

URL Extraction and Canonicalization

HTML parsers are freely available for many different languages. They provide the functionality to easily identify HTML tags and associated attribute–value pairs in a given HTML document. In order to extract hyperlink URLs from a Web page, we can use these parsers to find anchor tags and grab the values of associated `href` attributes. However, we do need to convert any relative URLs to absolute URLs using the base URL of the page from where they were retrieved.

Different URLs that correspond to the same Web page can be mapped onto a single canonical form. This is important in order to avoid fetching the same page many times. Here are some of the steps used in typical URL canonicalization procedures:

- Convert the protocol and hostname to lowercase. For example, `HTTP://www.UIOWA.edu` is converted to `http://www.uiowa.edu`.
- Remove the "anchor" or "reference" part of the URL. Hence, `http://myspiders.biz.uiowa.edu/faq.html#what` is reduced to `http://myspiders.biz.uiowa.edu/faq.html`.
- Perform URL encoding for some commonly used characters such as "~". This would prevent the crawler from treating `http://dollar.biz.uiowa.edu/~pant/` as a different URL from `http://dollar.biz.uiowa.edu/%7Epant/`.
- For some URLs, add trailing "/"s. `http://dollar.biz.uiowa.edu` and `http://dollar.biz.uiowa.edu/` must map to the same canonical form. The decision to add a trailing "/" will require heuristics in many cases.
- Use heuristics to recognize default Web pages. File names such as `index.html` or `index.htm` may be removed from the URL with the assumption that they are the default files. If that is true, they would be retrieved by simply using the base URL.
- Remove ".." and its parent directory from the URL path. Therefore, URL path `/%7Epant/BizIntel/Seeds/../ODPSeeds.dat` is reduced to `/%7Epant/BizIntel/ODPSeeds.dat`.
- Leave the port numbers in the URL unless it is port 80. As an alternative, leave the port numbers in the URL and add port 80 when no port number is specified.

It is important to be consistent while applying canonicalization rules. It is possible that two seemingly opposite rules work equally well (such as that for port numbers)

as long as you apply them consistently across URLs. Other canonicalization rules may be applied based on the application and prior knowledge about some sites (e.g., known mirrors).

As noted earlier spider traps pose a serious problem for a crawler. The "dummy" URLs created by spider traps often become increasingly larger in size. A way to tackle such traps is by limiting the URL sizes to, say, 128 or 256 characters.

Stoplisting and Stemming

When parsing a Web page to extract content information or in order to score new URLs suggested by the page, it is often helpful to remove commonly used words or *stopwords*[6] such as "it" and "can". This process of removing stopwords from text is called *stoplisting*. Note that the Dialog[7] system recognizes no more than nine words ("an," "and," "by," "for," "from," "of," "the," "to," and "with") as the stopwords. In addition to stoplisting, one may also stem the words found in the page. The *stemming* process normalizes words by conflating a number of morphologically similar words to a single root form or stem. For example, "connect," "connected," and "connection" are all reduced to "connect." Implementations of the commonly used Porter stemming algorithm [29] are easily available in many programming languages. One of the authors has experienced cases in the biomedical domain where stemming reduced the precision of the crawling results.

2.5 HTML tag tree

Crawlers may assess the value of a URL or a content word by examining the HTML tag context in which it resides. For this, a crawler may need to utilize the tag tree or Document Object Model (DOM) structure of the HTML page [8, 24, 27]. Figure 2 shows a tag tree corresponding to an HTML source. The `<html>` tag forms the root of the tree, and various tags and texts form nodes of the tree. Unfortunately, many Web pages contain badly written HTML. For example, a start tag may not have an end tag (it may not be required by the HTML specification), or the tags may not be properly nested. In many cases, the `<html>` tag or the `<body>` tag is altogether missing from the HTML page. Thus structure-based criteria often require the prior step of converting a "dirty" HTML document into a well-formed one, a process that is called *tidying* an HTML page.[8] This includes both the insertion of missing tags and the reordering of tags in the page. Tidying an HTML page is necessary for mapping the content of a page onto a tree structure with integrity, where each node has a single parent. Hence, it is an essential precursor to analyzing an HTML page as a tag tree. Note that analyzing the DOM structure is only necessary if the topical crawler intends to use the HTML document structure in a nontrivial manner. For example, if

---

[6] for an example list of stopwords refer to `http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words`

[7] http://www.dialog.com

[8] `http://www.w3.org/People/Raggett/tidy/`

the crawler only needs the links within a page, and the text or portions of the text in the page, one can use simpler HTML parsers. Such parsers are also readily available in many languages.
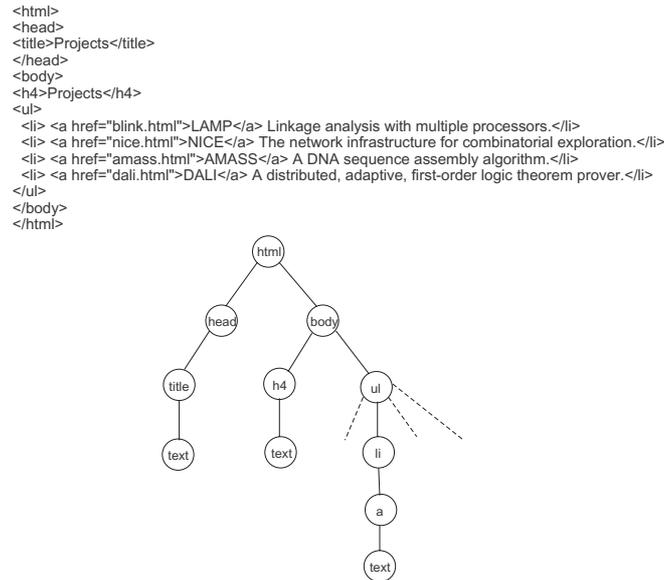
```
<html>
<head>
<title>Projects</title>
</head>
<body>
<h4>Projects</h4>
<ul>
 <li> <a href="blink.html">LAMP</a> Linkage analysis with multiple processors.</li>
 <li> <a href="nice.html">NICE</a> The network infrastructure for combinatorial exploration.</li>
 <li> <a href="amass.html">AMASS</a> A DNA sequence assembly algorithm.</li>
 <li> <a href="dali.html">DALI</a> A distributed, adaptive, first-order logic theorem prover.</li>
</ul>
</body>
</html>
```

Fig. 2. An HTML page and the corresponding tag tree

2.6 Multithreaded Crawlers

A sequential crawling loop spends a large amount of time in which either the CPU is idle (during network/disk access) or the network interface is idle (during CPU operations). Multithreading, where each thread follows a crawling loop, can provide reasonable speed-up and efficient use of available bandwidth. Figure 3 shows a multithreaded version of the basic crawler in Fig. 1. Note that each thread starts by locking the frontier to pick the next URL to crawl. After picking a URL it unlocks the frontier allowing other threads to access it. The frontier is again locked when new URLs are added to it. The locking steps are necessary in order to synchronize the use of the frontier that is now shared among many crawling loops (threads). The model of multithreaded crawler in Fig. 3 follows a standard parallel computing model [18]. Note that a typical crawler would also maintain a shared history data structure for a fast lookup of URLs that have been crawled. Hence, in addition to the frontier it would also need to synchronize access to the history.

The multithreaded crawler model needs to deal with an empty frontier just like a sequential crawler. However, the issue is less simple now. If a thread finds the

frontier empty, it does not automatically mean that the crawler as a whole has reached a dead end. It is possible that other threads are fetching pages and may add new URLs in the near future. One way to deal with the situation is by sending a thread to a sleep state when it sees an empty frontier. When the thread wakes up, it checks again for URLs. A global monitor keeps track of the number of threads currently sleeping. Only when all the threads are in the sleep state does the crawling process stop. More optimizations can be performed on the multithreaded model described here, as for instance to decrease contentions between the threads and to streamline network access.
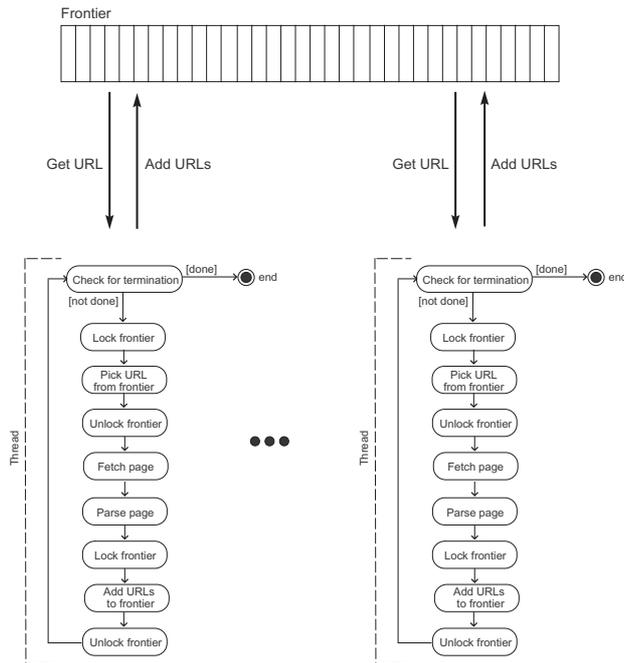

Fig. 3. A multithreaded crawler model

This section described the general components of a crawler. The common infrastructure supports at one extreme a very simple breadth-first crawler and at the other end crawler algorithms that may involve very complex URL selection mechanisms. Factors such as frontier size, page parsing strategy, crawler history, and page repository have been identified as interesting and important dimensions to crawler definitions.

3 Crawling Algorithms

We now discuss a number of crawling algorithms that are suggested in the literature. Note that many of these algorithms are variations of the best-first scheme. The difference is in the heuristics they use to score the unvisited URLs, with some algorithms adapting and tuning their parameters before or during the crawl.

3.1 Naive Best-First Crawler

A naive best-first was one of the crawlers detailed and evaluated by the authors in an extensive study of crawler evaluation [22]. This crawler represents a fetched Web page as a vector of words weighted by occurrence frequency. The crawler then computes the cosine similarity of the page to the query or description provided by the user, and scores the unvisited URLs on the page by this similarity value. The URLs are then added to a frontier that is maintained as a priority queue based on these scores. In the next iteration each crawler thread picks the best URL in the frontier to crawl, and returns with new unvisited URLs that are again inserted in the priority queue after being sco red based on the cosine similarity of the parent page. The cosine similarity between the page p and a query q is computed by:

$$sim(q, p) = \frac{\mathbf{v_q} \cdot \mathbf{v_p}}{\| \mathbf{v_q} \| \cdot \| \mathbf{v_p} \|}, \tag{1}$$

where $\mathbf{v_q}$ and $\mathbf{v_p}$ are term frequency (TF) based vector representations of the query and the page, respectively; $\mathbf{v_q} \cdot \mathbf{v_p}$ is the dot (inner) product of the two vectors; and $\| \mathbf{v} \|$ is the Euclidean norm of the vector $\mathbf{v}$. More sophisticated vector representation of pages, such as the TF-IDF [32] weighting scheme often used in information retrieval, are problematic in crawling applications because there is no a priori knowledge of the distribution of terms across crawled pages. In a multiple thread implementation the crawler acts like a best-N-first crawler where N is a function of the number of simultaneously running threads. Thus best-N-first is a generalized version of the best-first crawler that picks N best URLs to crawl at a time. In our research we have found the best-N-first crawler (with $N = 256$) to be a strong competitor [28, 23], showing clear superiority on the retrieval of relevant pages. Note that the best-first crawler keeps the frontier size within its upper bound by retaining only the best URLs based on the assigned similarity scores.

3.2 SharkSearch

SharkSearch [15] is a version of FishSearch [12] with some improvements. It uses a similarity measure like the one used in the naive best-first crawler for estimating the relevance of an unvisited URL. However, SharkSearch has a more refined notion of potential scores for the links in the crawl frontier. The anchor text, text surrounding the links or *link-context*, and inherited score from ancestors influence the potential scores of links. The ancestors of a URL are the pages that appeared on the crawl path

to the URL. SharkSearch, like its predecessor FishSearch, maintains a depth bound. That is, if the crawler finds unimportant pages on a crawl path it stops crawling farther along that path. To be able to track all the information, each URL in the frontier is associated with a depth and a potential score. The depth bound d is provided by the user, while the potential score of an unvisited URL is computed as:

$$score(url) = \gamma \cdot inherited(url) + (1 - \gamma) \cdot neighborhood(url), \qquad (2)$$

where $\gamma < 1$ is a parameter, the $neighborhood$ score signifies the contextual evidence found on the page that contains the hyperlink URL, and the $inherited$ score is obtained from the scores of the ancestors of the URL. More precisely, the $inherited$ score is computed as:

$$inherited(url) = \begin{cases} \delta \cdot sim(q, p); & \text{if } sim(q, p) > 0; \\ \delta \cdot inherited(p); & \text{otherwise}; \end{cases} \qquad (3)$$

where $\delta < 1$ is again a parameter, q is the query, and p is the page from which the URL was extracted.

The $neighborhood$ score uses the anchor text and the text in the "vicinity" of the anchor in an attempt to refine the overall $score$ of the URL by allowing for differentiation between links found within the same page. For that purpose, the SharkSearch crawler assigns an $anchor$ score and a $context$ score to each URL. The $anchor$ score is simply the similarity of the anchor text of the hyperlink containing the URL to the query q, i.e., $sim(q, anchor\_text)$. The $context$ score, on the other hand, broadens the context of the link to include some nearby words. The resulting augmented context $aug\_context$ is used for computing the $context$ score as follows:

$$context(url) = \begin{cases} 1; & \text{if } anchor(url) > 0; \\ sim(q, aug\_context); & \text{otherwise}. \end{cases} \qquad (4)$$

Finally, we derive the $neighborhood$ score from the $anchor$ score and the $context$ score as:

$$neighborhood(url) = \beta \cdot anchor(url) + (1 - \beta) \cdot context(url), \qquad (5)$$

where $\beta < 1$ is another parameter. We note that the implementation of SharkSearch would need to preset four different parameters d, γ, δ and β. Some values for the same are suggested by [15].

3.3 Focused Crawler

A focused crawler based on a hypertext classifier was developed by Chakrabarti et al. [9, 6]. The basic idea of the crawler was to classify crawled pages with categories in a topic taxonomy. To begin, the crawler requires a topic taxonomy such as Yahoo or the Open Directory Project (ODP).[9] In addition, the user provides example URLs

---

[9] `http://dmoz.org`

of interest (such as those in a bookmark file). The example URLs get automatically classified onto various categories of the taxonomy. Through an interactive process, the user can correct the automatic classification, add new categories to the taxonomy, and mark some of the categories as "good" (i.e., of interest to the user). The crawler uses the example URLs to build a Bayesian classifier that can find the probability $(\Pr(c|p))$ that a crawled page p belongs to a category c in the taxonomy. Note that by definition $\Pr(r|p) = 1$, where r is the root category of the taxonomy. A relevance score associated with each crawled page is computed as:

$$R(p) = \sum_{c \in good} \Pr(c|p). \tag{6}$$

When the crawler is in a "soft" focused mode, it uses the relevance score of the crawled page to score the unvisited URLs extracted from it. The scored URLs are then added to the frontier. Then in a manner similar to the naive best-first crawler, it picks the best URL to crawl next. In the "hard" focused mode, for a crawled page p, the classifier first finds the leaf node c^* (in the taxonomy) with maximum probability of including p. If any of the parents (in the taxonomy) of c^* are marked as ' 'good" by the user, then the URLs from the crawled page p are extracted and added to the frontier.

Another interesting element of the focused crawler is the use of a distiller. The distiller applies a modified version of Kleinberg's algorithm [17] to find topical *hubs*. The hubs provide links to many authoritative sources on the topic. The distiller is activated at various times during the crawl and some of the top hubs are added to the frontier.

3.4 Context Focused Crawler

Context focused crawlers [13] use Bayesian classifiers to guide their crawl. However, unlike the focused crawler described above, these classifiers are trained to estimate the link distance between a crawled page and the relevant pages. We can appreciate the value of such an estimation from our own browsing experiences. If we are looking for papers on "numerical analysis," we may first go to the home pages of math or computer science departments and then move to faculty pages, which may then lead to the relevant papers. A math department Web site may not have the words "numerical analysis" on its home page. A crawler such as the naive best-first crawler would put such a page on low priority and might never visit it. However, if the crawler could estimate that a relevant paper on "numerical analysis" is probably two links away, we would have a way of giving the home page of the math department higher priority than the home page of a law school.

The context focused crawler is trained using a *context graph* of L layers corresponding to each seed page. The seed page forms layer 0 of the graph. The pages corresponding to the in-links to the seed page are in layer 1. The in-links to the layer 1 pages make up the layer 2 pages, and so on. We can obtain the in-links to pages of any layer by using a search engine. Figure 4 depicts a context graph for

`http://www.biz.uiowa.edu/programs/` as seed. Once the context graphs for all of the seeds are obtained, the pages from the same layer (number) from each graph are combined into a single layer. This gives a new set of layers of what is called a *merged context graph*. This is followed by a feature selection stage where the seed pages (or possibly even layer 1 pages) are concatenated into a single large document. Using the TF-IDF [32] scoring scheme, the top few terms are identified from this document to represent the vocabulary (feature space) that will be used for classification.

A set of naive Bayes classifiers are built, one for each layer in the merged context graph. All the pages in a layer are used to compute $\Pr(t|c_l)$, the probability of occurrence of a term t given the class c_l corresponding to layer l. A prior probability, $\Pr(c_l) = 1/L$, is assigned to each class where L is the number of layers. The probability of a given page p belonging to a class c_l can then be computed as $\Pr(c_l|p)$. Such probabilities are computed for each class. The class with highest probability is treated as the winning class (layer). However, if the probability for the winning class is still less than a threshold, the crawled page is classified into the "other" class. This "other" class represents pages that do not have a good fit with any of the classes of the context graph. If the probability of the winning class does exceed the threshold, the page is classified into the winning class.

The set of classifiers corresponding to the context graph provides us with a mechanism to estimate the link distance of a crawled page from relevant pages. If the mechanism works, the math department home page will get classified into layer 2, while the law school home page will get classified to "others." The crawler maintains a queue for each class, containing the pages that are crawled and classified into that class. Each queue is sorted by the the probability scores $\Pr(c_l|p)$. When the crawler needs a URL to crawl, it picks the top page in the nonempty queue with smallest l. So it will tend to pick up pages that seem to be closer to the relevant pages first. The out-links from such pages will get explored before the out-links of pages that seem to be far away from the relevant portions of the Web.

3.5 InfoSpiders

In InfoSpiders [21, 23], an adaptive population of agents searches for pages relevant to the topic. Each agent is essentially following the crawling loop (Sect. 2) while using an adaptive query list and a neural net to decide which links to follow. The algorithm provides an exclusive frontier for each agent. In a multithreaded implementation of InfoSpiders (see Sect. 5.1) each agent corresponds to a thread of execution. Hence, each thread has a noncontentious access to its own frontier. Note that any of the algorithms described in this chapter may be implemented similarly (one frontier per thread). In the original algorithm (e.g., [21]) each agent kept its frontier limited to the links on the page that was last fetched by the agent. As a result of this limited memory approach the crawler was limited to following the links on the current page, and it was outperformed by the naive best-first crawler on a number of evaluation criteria [22]. Since then a number of improvements (inspired by naive best-first) to the original algorithm have been designed while retaining its capability to learn
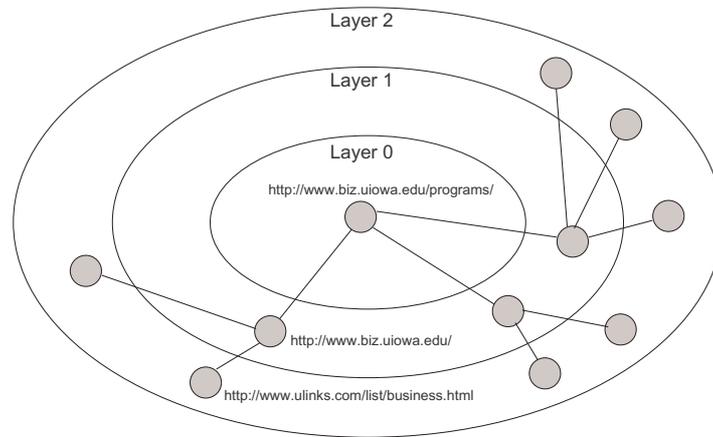
Fig. 4. A context graph

link estimates via neural nets and focus its search toward more promising areas by selective reproduction. In fact, the redesigned version of the algorithm has been found to outperform various versions of naive best-first crawlers on specific crawling tasks with crawls that are longer than ten thousand pages [23].

The adaptive representation of each agent consists of a list of keywords (initialized with a query or description) and a neural net used to evaluate new links. Each input unit of the neural net receives a count of the frequency with which the keyword occurs in the vicinity of each link to be traversed, weighted to give more importance to keywords occurring near the link (and maximum in the anchor text). There is a single output unit. The output of the neural net is used as a numerical quality estimate for each link considered as input. These estimates are then combined with estimates based on the cosine similarity (Eq. (1)) between the agent's keyword vector and the page containing the links. A parameter $\alpha, 0 \leq \alpha \leq 1$, regulates the relative importance given to the estimates based on the neural net versus the parent page. Based on the combined score, the agent uses a stochastic selector to pick one of the links in the frontier with probability

$$\Pr(\lambda) = \frac{e^{\beta\sigma(\lambda)}}{\sum_{\lambda' \in \phi} e^{\beta\sigma(\lambda')}}, \tag{7}$$

where λ is a URL in the local frontier ϕ and $\sigma(\lambda)$ is its combined score. The β parameter regulates the greediness of the link selector.

After a new page has been fetched, the agent receives "energy" in proportion to the similarity between its keyword vector and the new page. The agent's neural net can be trained to improve the link estimates by predicting the similarity of the new page, given the inputs from the page that contained the link leading to it. A back-propagation algorithm is used for learning. Such a learning technique provides InfoSpiders with the unique capability to adapt the link-following behaviour in the course of a crawl

by associating relevance estimates with particular patterns of keyword frequencies around links.

An agent's energy level is used to determine whether or not an agent should reproduce after visiting a page. An agent reproduces when the energy level passes a constant threshold. The reproduction is meant to bias the search toward areas (agents) that lead to good pages. At reproduction, the offspring (new agent or thread) receives half of the parent's link frontier. The offspring's keyword vector is also mutated (expanded) by adding the term that is most frequent in the parent's current document. This term addition strategy in a limited way is comparable to the use of classifiers in Sect. 3.4, since both try to identify lexical cues that appear on pages leading up to the relevant pages.

In this section we have presented a variety of crawling algorithms, most of which are variations of the best-first scheme. The readers may pursue Menczer et. al. [23] for further details on the algorithmic issues related with some of the crawlers.

4 Evaluation of Crawlers

In a general sense, a crawler (especially a topical crawler) may be evaluated on its ability to retrieve "good" pages. However, a major hurdle is the problem of recognizing these good pages. In an operational environment real users may judge the relevance of pages as these are crawled, allowing us to determine if the crawl was successful or not. Unfortunately, meaningful experiments involving real users for assessing Web crawls are extremely problematic. For instance, the very scale of the Web suggests that in order to obtain a reasonable notion of crawl effectiveness one must conduct a large number of crawls, i.e., involve a large number of users.

Second, crawls against the live Web pose serious time constraints. Therefore crawls other than short-lived ones will seem overly burdensome to the user. We may choose to avoid these time loads by showing the user the results of the full crawl but this again limits the extent of the crawl.

In the not-so-distant future, the majority of the direct consumers of information is more likely to be Web agents working on behalf of humans and other Web agents than humans themselves. Thus it is quite reasonable to explore crawlers in a context where the parameters of crawl time and crawl distance may be beyond the limits of human acceptance imposed by user-based experimentation.

In general, it is important to compare topical crawlers over a large number of topics and tasks. This will allow us to ascertain the statistical significance of particular benefits that we may observe across crawlers. Crawler evaluation research requires an appropriate set of metrics. Recent research reveals several innovative performance measures. But first we observe that there are two basic dimensions in the assessment process, we need a measure of the crawled page's importance, and second we need a method to summarize performance across a set of crawled pages.

4.1 Page Importance

Let us enumerate some of the methods that have been used to measure page importance.

1. *Keywords in document*: A page is considered relevant if it contains some or all of the keywords in the query. Also, the frequency with which the keywords appear on the page may be considered [10].
2. *Similarity to a query*: Often a user specifies an information need as a short query. In some cases a longer description of the need may be available. Similarity between the short or long description and each crawled page may be used to judge the page's relevance [15, 22].
3. *Similarity to seed pages*: The pages corresponding to the seed URLs are used to measure the relevance of each page that is crawled [2]. The seed pages are combined together into a single document and the cosine similarity of this document and a crawled page is used as the page's relevance score.
4. *Classifier score*: A classifier may be trained to identify the pages that are relevant to the information need or task. The training is done using the seed (or prespecified relevant) pages as positive examples. The trained classifier will then provide Boolean or continuous relevance scores to each of the crawled pages [9, 13].
5. *Retrieval system rank*: N different crawlers are started from the same seeds and allowed to run until each crawler gathers P pages. All of the $N \cdot P$ pages collected from the crawlers are ranked against the initiating query or description using a retrieval system such as SMART. The rank provided by the retrieval system for a page is used as its relevance score [22].
6. *Link-based popularity*: One may use algorithms, such as PageRank [5] or Hyperlink-Induced Topic Search (HITS) [17], that provide popularity estimates of each of the crawled pages. A simpler method would be to use just the number of in-links to the crawled page to derive similar information [10, 2]. Many variations of link-based methods using topical weights are choices for measuring topical popularity of pages [4, 7].

4.2 Summary Analysis

Given a particular measure of page importance we can summarize the performance of the crawler with metrics that are analogous to the information retrieval (IR) measures of *precision* and *recall*. Precision is the fraction of retrieved (crawled) pages that are relevant, while recall is the fraction of relevant pages that are retrieved (crawled). In a usual IR task the notion of a relevant set for recall is restricted to a given collection or database. Considering the Web to be one large collection, the relevant set is generally unknown for most Web IR tasks. Hence, explicit recall is hard to measure. Many authors provide precision-like measures that are easier to compute in order to evaluate the crawlers. We will discuss a few such precision-like measures:

1. *Acquisition rate*: In cases where we have Boolean relevance scores we could measure the explicit rate at which "good" pages are found. Therefore, if 50

relevant pages are found in the first 500 pages crawled, then we have an acquisition rate or *harvest rate* [1] of 10% at 500 pages.

2. *Average relevance*: If the relevance scores are continuous they can be averaged over the crawled pages. This is a more general form of harvest rate [9, 22, 8]. The scores may be provided through simple cosine similarity or a trained classifier. Such averages (Fig. 6(a)) may be computed over the progress of the crawl (first 100 pages, first 200 pages, and so on) [22]. Sometimes running averages are calculated over a window of a few pages (e.g., the last 50 pages from a current crawl point) [9].

Since measures analogous to recall are hard to compute for the Web, authors resort to indirect indicators for estimating recall. Some such indicators are:

1. *Target recall*: A set of known relevant URLs is split into two disjoint sets–*targets* and *seeds*. The crawler is started from the seeds pages and the recall of the targets is measured. The target recall is computed as

$$target\_recall = \frac{|\ \mathcal{P}_t \cap \mathcal{P}_c\ |}{|\ \mathcal{P}_t\ |}$$

, where $\mathcal{P}_t$ is the set of target pages, and $\mathcal{P}_c$ is the set of crawled pages. The recall of the target set is used as an estimate of the recall of relevant pages. Figure 5 gives a schematic justification of the measure. Note that the underlying assumption is that the targets are a random subset of the relevant pages.
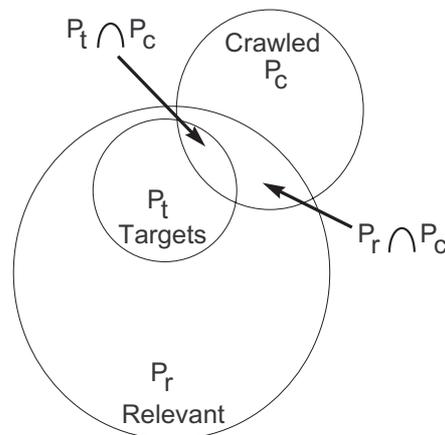

Fig. 5. The performance metric $|\ \mathcal{P}_t \cap \mathcal{P}_c\ |\ /\ |\ \mathcal{P}_t\ |$ as an estimate of $|\ \mathcal{P}_r \cap \mathcal{P}_c\ |\ /\ |\ \mathcal{P}_r\ |$

2. *Robustness*: The seed URLs are split into two disjoint sets S_a and S_b. Each set is used to initialize an instance of the same crawler. The overlap in the pages crawled starting from the two disjoint sets is measured. A large overlap is interpreted as *robustness* of the crawler in covering relevant portions of the Web [9, 6].

There are other metrics that measure the crawler performance in a manner that combines both precision and recall. For example, *search length* [21] measures the number of pages crawled before a certain percentage of the relevant pages are retrieved.

Figure 6 shows an example of performance plots for two different crawlers. The crawler performance is depicted as a trajectory over time (approximated by crawled pages). The naive best-first crawler is found to outperform the breadth-first crawler based on evaluations over 159 topics with 10,000 pages crawled by each crawler on each topic (hence the evaluation involves millions of pages).

In this section we have outlined methods for assessing page importance and measures to summarize crawler performance. When conducting a fresh crawl experiment it is important to select an evaluation approach that provides a reasonably complete and sufficiently detailed picture of the crawlers being compared.

5 Applications

We now briefly review a few applications that use crawlers. Our intent is not to be comprehensive but instead to simply highlight their utility.

5.1 MySpiders: Query-Time Crawlers

MySpiders [26] is a Java applet that implements the InfoSpiders and the naive best-first algorithms. Multithreaded crawlers are started when a user submits a query. Results are displayed dynamically as the crawler finds "good" pages. The user may browse the results while the crawling continues in the background. The multithreaded implementation of the applet deviates from the general model specified in Fig. 3. In line with the autonomous multiagent nature of the InfoSpiders algorithm (Sect.3.5), each thread has a separate frontier. This applies to the naive best-first algorithm as well. Hence, each thread is more independent with noncontentious access to its frontier. The applet allows the user to specify the crawling algorithm and the maximum number of pages to fetch. In order to initiate the crawl, the system uses the Google Web API[10] to obtain a few seed pages. The crawler threads are started from each of the seeds, and the crawling continues until the required number of pages are fetched or the frontier is empty. Figure 7 shows MySpiders working on a user query using the InfoSpiders algorithm.

5.2 CORA: Building Topic-Specific Portals

A topical crawler may be used to build topic-specific portals such as sites that index research papers. One such application developed by McCallum et al. [20] collected and maintained research papers in Computer Science (CORA). The crawler used by the application is based on reinforcement learning (RL) that allows for finding
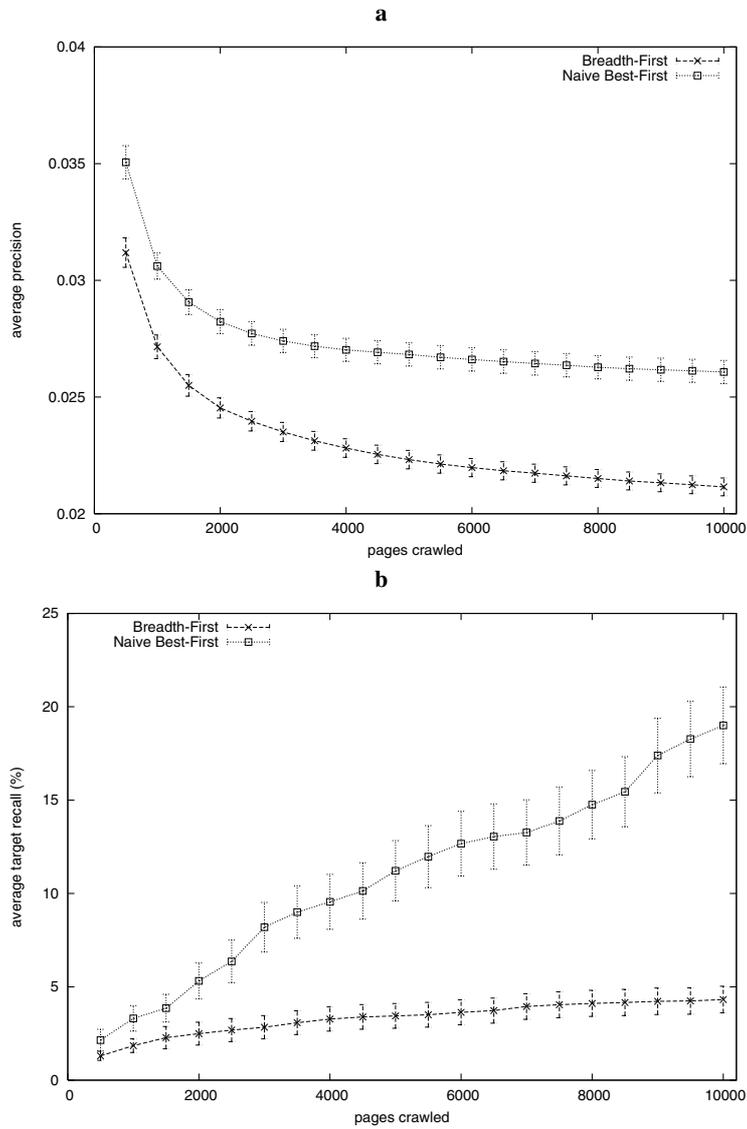
---

[10] http://www.google.com/apis

a

b

Fig. 6. Performance Plots: (**a**) average precision (similarity to topic description); (**b**) average target recall. The averages are calculated over 159 topics and the error bars show ± 1 standard error. One-tailed t-test for the alternative hypothesis that the naive best-first crawler outperforms the breadth-first crawler (at 10,000 pages) generates p values that are < 0.01 for both performance metrics
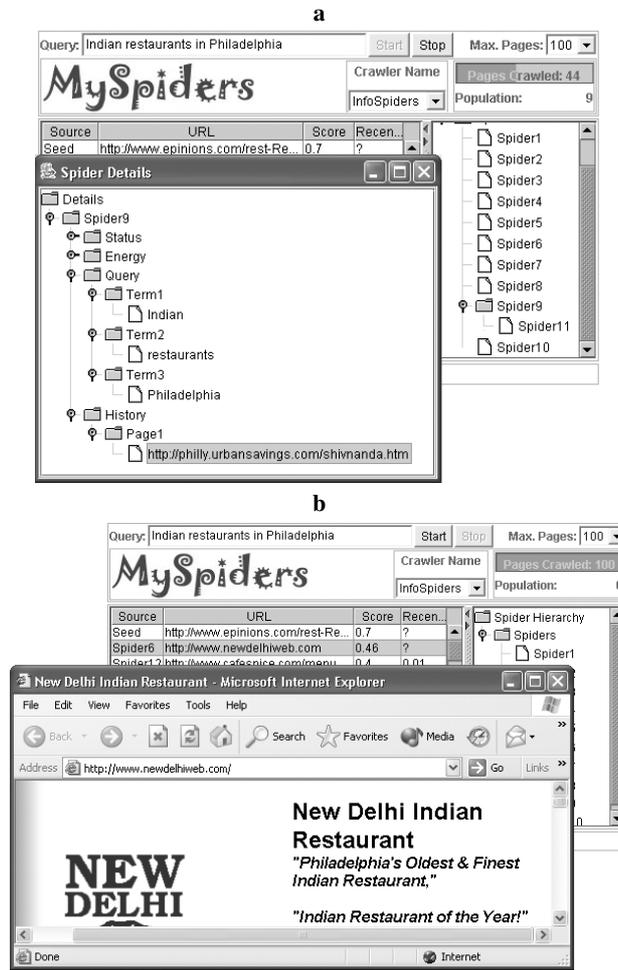
Fig. 7. The user interface of *MySpiders* during a crawl using the InfoSpiders algorithm. (**a**) In the search process, Spider 9 has reproduced and its progeny is visible in the expandable tree (*right*). A spider's details are revealed by clicking on it on the tree (*left*). (**b**) At the end of the crawl, one of the top hits is found by a spider (and it is not one of the seeds). The hit is viewed by clicking its URL in the results frame

crawling policies that lead to immediate as well as long -term benefits. The benefits are discounted based on how far away they are from the current page. Hence, a hyperlink that is expected to immediately lead to a relevant page is preferred over one that is likely to bear fruit after a few links. The need to consider future benefit along a crawl path is motivated by the fact that lexical similarity between pages falls rapidly with increasing link distance. Therefore, as noted earlier, a math department home page that leads to a numerical analysis paper may provide very little lexical signal

to a naive best-first crawler that is searching for the paper. Hence, the motivation of the RL crawling algorithm is similar to that of the context focused crawler. The RL crawler was trained using known paths to relevant pages. The trained crawler is then use to estimate the benefit of following a hyperlink.

5.3 Mapuccino: Building Topical Site Maps

One approach to building site maps is to start from a seed URL and crawl in a breadth-first manner until a certain number of pages have been retrieved or a certain depth has been reached. The site map may then be displayed as a graph of connected pages. However, if we are interested in building a site map that focuses on a certain topic, then the above-mentioned approach will lead a to a large number of unrelated pages as we crawl to greater depths or fetch more pages. Mapuccino [15] corrects this by using SharkSearch (Sect. 3.2) to guide the crawler and then build a visual graph that highlights the relevant pages.

5.4 Letizia: a Browsing Agent

Letizia [19] is an agent that assists a user during browsing. While the user surfs the Web, Letizia tries to understand user interests based on the pages being browsed. The agent then follows the hyperlinks starting from the current page being browsed to find pages that could be of interest to the user. The hyperlinks are crawled automatically and in a breadth-first manner. The user is not interrupted, but pages of possible interest are suggested only when she needs recommendations. The agent makes use of topical locality on the Web [11] to provide context-sensitive results.

5.5 Other Applications

Crawling in general and topical crawling in particular is being applied for various other applications, many of which do not appear as technical papers. For example, business intelligence has much to gain from topical crawling. A large number of companies have Web sites where they often describe their current objectives, future plans, and product lines. In some areas of business, there are a large number of start-up companies that have rapidly changing Web sites. All these factors make it important for various business entities to use sources other than the general-purpose search engines to keep track of relevant and publicly available information about their potential competitors or collaborators [27].

Crawlers have also been used for biomedical applications like finding relevant literature on a gene [33]. On a different note, there are some controversial applications of crawlers such as extracting e-mail addresses from Web sites for spamming.

6 Conclusion

Because of the dynamism of the Web, crawling forms the backbone of applications that facilitate Web information retrieval. While the typical use of crawlers has been for

creating and maintaining indexes for general-purpose search engines, diverse usage of topical crawlers is emerging both for client and server-based applications. Topical crawlers are becoming important tools to support applications such as specialized Web portals, online searching, and competitive intelligence. A number of topical crawling algorithms have been proposed in the literature. Often the evaluation of these crawlers is done by comparing a few crawlers on a limited number of queries/tasks without considerations of statistical significance. Anecdotal results, while important, do not suffice for thorough performance comparisons. As the Web crawling field matures, the disparate crawling strategies will have to be evaluated and compared on common tasks through well-defined performance measures.

In the future, we see more sophisticated usage of hypertext structure and link analysis by the crawlers. For a current example, Chakrabarti et. al. [8] have suggested the use of the pages' HTML tag tree or DOM structure for focusing a crawler. While they have shown some benefit of using the DOM structure, a thorough study on the merits of using the structure (in different ways) for crawling is warranted [24]. Topical crawlers depend on various cues from crawled pages to prioritize the fetching of unvisited URLs. A good understanding of the relative importance of cues such as the link context, linkage (graph) structure, ancestor pages, and so on is also needed [16]. Another potential area of research is stronger collaboration between search engines and crawlers [25], and among the crawlers themselves. The scalability benefits of distributed topical crawling [9, 21] are yet to be fully realized. Can crawlers help a search engine to focus on user interests? Can a search engine help a crawler to focus on a topic? Can a crawler on one machine help a crawler on another? Many such questions will motivate future research and crawler applications.

Acknowledgments

References

1. C. C. Aggarwal, F. Al-Garawi, and P. S. Yu. Intelligent crawling on the World Wide Web with arbitrary predicates. In *WWW10*, Hong Kong, May 2001.
2. B. Amento, L. Terveen, and W. Hill. Does "authority" mean quality? Predicting expert quality ratings of web documents. In *Proc. 23th Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, Athens, Greece, 2000.
3. A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the Web. *ACM Transactions on Internet Technology*, 1(1), 2001.
4. K. Bharat and M.R. Henzinger. Improved algorithms for topic distillation in a hyperlinked environment. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Melbourne, Australia, 1998.
5. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30:107–117, 1998.

6. S. Chakrabarti. *Mining the Web*. Morgan Kaufmann, 2003.

7. S. Chakrabarti, B. Dom, D. Gibson, J. Kleinberg, P. Raghavan, and S. Rajagopalan. Automatic resource list compilation by analyzing hyperlink structure and associated text. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, 1998.

8. S. Chakrabarti, K. Punera, and M. Subramanyam. Accelerated focused crawling through online relevance feedback. In *WWW2002*, Hawaii, May 2002.

9. S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific Web resource discovery. *Computer Networks*, 31(11–16):1623–1640, 1999.

10. J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. *Computer Networks*, 30:161–172, 1998.

11. B.D. Davison. Topical locality in the web. In *Proc. 23rd Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, Athens, Greece, 2000.

12. P. M. E. De Bra and R. D. J. Post. Information retrieval in the World Wide Web: Making client-based searching feasible. In *Proc. 1st International World Wide Web Conference*, 1994.

13. M. Diligenti, F. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused crawling using context graphs. In *Proc. 26th International Conference on Very Large Databases (VLDB 2000)*, pages 527–534, Cairo, Egypt, 2000.

14. D. Eichmann. Ethical Web agents. In *Second International World-Wide Web Conference*, pages 3–13, Chicago, Illinois, 1994.

15. M. Hersovici, M. Jacovi, Y. S. Maarek, D. Pelleg, M. Shtalhaim, and S. Ur. The shark-search algorithm — An application: Tailored Web site mapping. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, 1998.

16. J. Johnson, K. Tsioutsiouliklis, and C.L. Giles. Evolving strategies for focused web crawling. In *Proc. 12th Intl. Conf. on Machine Learning (ICML-2003)*, Washington DC, 2003.

17. J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

18. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

19. H. Lieberman, F. Christopher, and L. Weitzman. Exploring the Web with reconnaissance agents. *Communications of the ACM*, 44:69–75, August 2001.

20. A.K. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.

21. F. Menczer and R. K. Belew. Adaptive retrieval agents: Internalizing local context and scaling up to the Web. *Machine Learning*, 39(2–3):203–242, 2000.

22. F. Menczer, G. Pant, M. Ruiz, and P. Srinivasan. Evaluating topic-driven Web crawlers. In *Proc. 24th Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, New Orleans, Louisiana, 2001.

23. F. Menczer, G. Pant, and P. Srinivasan. Topical web crawlers: Evaluating adaptive algorithms. *To appear in ACM Trans. on Internet Technologies*, 2003. http://dollar.biz.uiowa.edu/˜fil/Papers/TOIT.pdf.

24. G. Pant. Deriving link-context from HTML tag tree. In *8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2003.

25. G. Pant, S. Bradshaw, and F. Menczer. Search engine-crawler symbiosis: Adapting to community interests. In *Proc. 7th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2003)*, Trondheim, Norway, 2003.

26. G. Pant and F. Menczer. MySpiders: Evolve your own intelligent Web crawlers. *Autonomous Agents and Multi-Agent Systems*, 5(2):221–229, 2002.

27. G. Pant and F. Menczer. Topical crawling for business intelligence. In *Proc. 7th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2003)*, Trondheim, Norway, 2003.

28. G. Pant, P. Srinivasan, and F. Menczer. Exploration versus exploitation in topic driven crawlers. In *WWW02 Workshop on Web Dynamics*, Honolulu, Hawaii, 2002.

29. M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

30. S. RaviKumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the Web graph. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 57–65, Redondo Beach, CA, Nov. 2000.

31. J. Rennie and A. K. McCallum. Using reinforcement learning to spider the Web efficiently. In *Proc. 16th International Conf. on Machine Learning*, pages 335–343, Bled, Slovenia, 1999.

32. G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY, 1983.

33. P. Srinivasan, J. Mitchell, O. Bodenreider, G. Pant, and F. Menczer. Web crawling agents for retrieving biomedical information. In *NETTAB: Agents in Bioinformatics*, Bologna, Italy, 2002.