Adaptive Web Crawling through Structure-Based Link Classification

Muhammad Faheem[1,2] and Pierre Senellart[1,3]

[1] LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay, Paris, France
[2] University of Ottawa, Ottawa, Canada
[3] IPAL, CNRS, National University of Singapore, Singapore

Abstract. Generic web crawling approaches cannot distinguish among various page types and cannot target content-rich areas of a website. We study the problem of efficient unsupervised web crawling of content-rich webpages. We propose ACEBot (Adaptive Crawler Bot for data Extraction), a structure-driven crawler that uses the inner structure of the pages and guides the crawling process based on the importance of their content. ACEBot works in two phases: in the *learning* phase, it constructs a dynamic site map (limiting the number of URLs retrieved) and learns a traversal strategy based on the importance of *navigation patterns* (selecting those leading to valuable content); in the *intensive crawling* phase, ACEBot performs massive downloading following the chosen navigation patterns. Experiments over a large dataset illustrate the effectiveness of our system.

1 Introduction

A large part of web content is found on websites powered by content management systems (CMSs) such as vBulletin, phpBB, or WordPress [1]. The presentation layer of these CMSs use predefined templates (which may include left or right sidebars, headers and footers, navigation bars, main content, etc.) for populating the content of the requested web document from an underlying database. A study [1] found that 40–50% of web content, in 2005, was template-based; this order of magnitude is confirmed by a more recent web technology survey [2], which further shows that one specific CMS, WordPress, is used by 24% of websites, giving it 60% of CMS market share. Depending on the request, CMSs may use different templates for presenting information; e.g. in blogs, the *list of posts* type of page uses a different template than the *single post* webpage that also includes comments. These template-based webpages form a meaningful structure that mirrors the implicit logical relationship between web content across different pages within a website. Many templates are used by CMSs for generating different types of webpages. Each template generates a set of webpages (e.g. list of blog posts) that share a common structure, but differ in content. These templates are consistently used across different regions of the site. More importantly, for a given template (say, a list of posts), the same links usually lead to the same kind of content (say, individual posts), with common layout and presentation properties.

Due to limited bandwidth, storage, or indexing capabilities, only a small fraction of web content can actually be harvested by web crawlers. This is true for archival crawlers of institutions with limited resources (e.g. the national library of a small country). This

is even true for a company such as Google, that, as of June 2013, had discovered more than a trillion unique URLs [3], but indexed around 40 billion webpages. This suggests a need to develop a crawling strategy that not only effectively crawls web content from template-based websites, but also efficiently minimizes the number of HTTP requests by avoiding non-interesting webpages.

A generic web crawler performs inefficient crawling of websites. It crawls the web with no guarantee of content quality. An ideal crawling approach should solve the following three problems: What kind of webpages are *important* to crawl (to avoid redundant and invalid pages)? Which *important* links should be followed? What *navigation patterns* are required on the website?

We introduce in this article an intelligent crawling technique that meets these criteria. We propose a *structure-driven* approach that is more precise, effective, and achieves a higher quality level, without loss of information. It guides the crawler towards content-rich areas: this is achieved by learning the best traversal strategy (a collection of important navigation patterns) during a learning phase, that ultimately guides the crawler to crawl only content-rich webpages during an intensive crawling phase.

Our structure-driven crawler, ACEBot, first establishes connections among webpages based on their root-to-link paths in the DOM tree of the pages, then ranks paths according to their importance (i.e. root-to-links paths that lead to content-rich webpages), and further learns a traversal strategy for bulk-downloading of the website. Our main claim is that structure-based crawling not only clusters webpages which require similar crawling actions, but also helps to identify duplicates, redundancy, and boilerplates, and plays as well an important role in prioritizing the *frontier* (the list of URLs left to crawl).

After discussing related work in Section 2, we present our model in Section 3. The algorithm that ACEBot follows is then presented in detail in Section 4, followed by experiments in Section 5. Due to space constraints, some material (proofs of results, additional experiments, examples) could not be included but can be found in [4].

2 Related work

In [5], Liu et al. proposed an algorithm, called *SEW*, that models a website as a hypertext structure. SEW relies on a combination of several domain-independent heuristics to identify the most important links within a webpage and thus discover a hierarchical organization of navigation and content pages. Kao et al. [6] have addressed a similar problem, and propose a technique to distinguish between pages containing *links* to news posts and the pages containing those news items. Compared to our approach, both techniques above only cluster the webpages into two predefined classes of pages: navigational and content pages. In addition, Kao et al. [6] focus on pages of a specific domain. In contrast, we have proposed a system that performs unsupervised crawling of websites (domain-independent) without prior assumption on the number of classes.

[7,8], aim to cluster webpages into different classes by exploiting their structural similarity at the DOM tree level, while [9] introduces crawling programs: a tool that listens to the user interaction, registers steps, and infers the corresponding intentional navigation. This approach is semi-supervised as it requires human interaction to learn navigation patterns to reach the content-rich pages. A web crawler is generally intended

for a massive crawling scenario, and thus semi-automatic approaches are not feasible in our setting. Therefore, in our proposed approach, we have introduced the learning phase which learns navigation patterns in an unsupervised manner.

Another structure-driven approach [10] has proposed a web crawler that requires minimum human effort. It takes a sample page and entry point as input and generates a set of navigation patterns (i.e. sequence of patterns) that guides a crawler to reach webpages structurally similar to the sample page. As stated above, this approach is also focused on a specific type of webpage, whereas our approach performs massive crawling at web scale for content-rich pages.

Several domain-dependent *web forum* crawling techniques [11,12] have been proposed recently. In [11], the crawler first clusters the webpages into two groups from a set of manual annotated pages using Support Vector Machines with some predefined features, and then, within each cluster, URLs are clustered using partial tree alignment. Furthermore, a set of ITF (index-thread-page-flipping) *regular expressions* are generated to launch a bulk download of a target web forum. The iRobot system [12], that we use as a baseline in our experiments, creates a sitemap of the website being crawled. The sitemap is constructed by randomly crawling a few webpages from a given website. After sitemap generation, iRobot obtains the structure of the web forum. The skeleton is obtained in the form of a directed graph consisting of vertices (webpages) and directed arcs (links between different webpages). A path analysis is then performed to learn an optimal traversal path which leads the extraction process in order to avoid redundant and invalid pages. A web-scale approach [13] has introduced an algorithm that performs URL-based clustering of webpages using some content features. However, in practice, URL-based clustering of webpages is less reliable in the presence of the dynamic nature of the web.

Our previous work [14] proposes an adaptive application-aware helper (AAH) that crawls known CMSs efficiently. AAH is assisted with a knowledge base that guides the crawling process. It first tries to detect the website and, if detected as a known one, attempts to identify the kind of webpage given the matched website. Then, the relevant crawling actions are executed for web archiving. This approach achieves the highest quality of web content with fewer HTTP requests, but is not fully automatic and requires a hand-written knowledge base that prevents crawling of unknown websites.

3 Model

In this section, we formalize our proposed model: we see the website to crawl as an directed graph, that is rooted (typically at the homepage of a site), and where edges are labeled (by structural properties of the corresponding hyperlink). We first consider the abstract problem, before explaining how actual websites fit into the model.

Formal definitions. We fix countable sets of *labels* $\mathscr{L}$ and *items* $\mathscr{I}$. Our main object of study is the graph to crawl:

Definition 1. *A* rooted graph *is a 5-tuple* $G = (V, E, r, \iota, l)$ *with* V *a finite set of vertices,* $E \subseteq V^2$ *a set of directed edges,* $r \in V$ *the* root*;* $\iota : V \to 2^{\mathscr{I}}$ *and* $l : E \to \mathscr{L}$ *assign respectively a set of* items *to every vertex and a* label *to every edge.*

Here, items serve to abstractly model the interesting content of webpages; the more items a crawl retrieves, the better. Labels are attached to hyperlinks between pages; further on, we will explain how we can use the DOM structure of a webpage to assign such labels. We naturally extend the function ι to a set of nodes X from G by posing: $\iota(X) := \bigcup_{u \in X} \iota(u)$. We introduce the standard notion of paths within the graph:

Definition 2. *Given a rooted graph $G = (V, E, r, \iota, l)$ and vertices $u, v \in V$, a* path *from u to v is a finite sequence of edges $e_1 \dots e_n$ from E such that there exists a set of nodes $u_1 \dots u_{n-1}$ in V with: $e_1 = (u, u_1)$; $\forall 1 < k < n, e_k = (u_{k-1}, u_k)$; $e_n = (u_{n-1}, v)$.*
The label *of the path $e_1 \dots e_n$ is the word $l(e_1) \dots l(e_n)$ over $\mathscr{L}$.*

Critical to our approach is the notion of a *navigation pattern* that uses edge labels to describe which paths to follow in a graph. Navigation patterns are defined using the standard automata-theory notion of *regular expression* (used here as *path expressions*):

Definition 3. *A* navigation pattern *p is a regular expression over $\mathscr{L}$. Given a graph $G = (V, E, r, \iota, l)$, the result of applying p onto G, denoted $p(G)$, is the set of nodes u with a path from r to u that has for label* a prefix of a word *in the language defined by p. We extend this notion to a finite set of navigation patterns P by letting $P(G) := \bigcup_{p \in P} p(G)$.*

Note that we require only a *prefix* of a word to match: a navigation pattern does not only return the set of pages whose path from the root matches the path expression, but also intermediate pages on those paths. For instance, consider a *path* $e_1 \dots e_n$ from r to a node u, such that the navigation pattern p is the path expression $l(e_1) \dots l(e_n)$. Then the result of executing navigation pattern p contains u, but also all pages on the path from r to u; more generally, p returns all pages whose path from the root matches a prefix of the expression $l(e_1) \dots l(e_n)$. Navigation patterns are assigned a score:

Definition 4. *Let $G = (V, E, r, \iota, l)$ be a rooted graph. The* score *of a finite set of navigation patterns P over G, denoted $\omega(P, G)$ is the average number of distinct items per node in $P(G)$: $\omega(P, G) := \frac{|\iota(P(G))|}{|P(G)|}$.*

In other words, a navigation pattern has a high score if it retrieves a large number of items in a relatively low number of nodes. The crawling interpretation is that we want to maximize the amount of useful content retrieved, while minimizing the number of HTTP requests made.

We can now formalize our problem of interest: given a rooted graph G and a collection of navigation patterns $\mathscr{P}$ (that may be all path expressions over $\mathscr{L}$ or a subclass of path expressions over $\mathscr{L}$), determine the set of navigation patterns $P \subseteq \mathscr{P}$ of maximal score over G. We can show (see [4] for proofs):

Proposition 1. *Given a graph G and a collection of navigation patterns $\mathscr{P}$, determining if one finite subset $P \subseteq \mathscr{P}$ has maximal score over G is a coNP-complete problem.*

Thus, there is no hope of efficiently obtaining an optimal set of navigation patterns. In this light we will introduce in Section 4 a non-optimal greedy approach to the selection of navigation patterns, that we will show in Section 5 still performs well in practice.

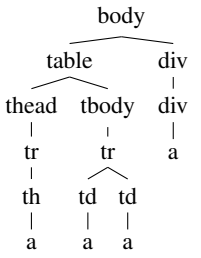

Fig. 1. Partial DOM tree representation

l_1 body/table/tbody/tr/td/a
l_2 body/table/thead/tr/th/a
l_3 body/div/div/a

Fig. 2. Root-to-link paths

NP	#2-grams	score
l_1	5107	2553.5
$l_1 l_4$	7214	2404.7
l_3	754	754.0
l_2	239	239.0

Fig. 3. Navigation patterns with score

Model generation. We now explain how we consider crawling a website in the previously introduced abstract model. A website is any HTTP-based application, formed with a set of interlinked webpages that can be traversed from some base URL, such as `http://icadl2015.org/`. The base URL of a website is called the entry point of the site. For our purpose, we model a given website as a directed graph (see Definition 1), where the base URL becomes the root of the graph. Each vertex of the graph represents a distinct webpage and, following Definition 1, a set of items is assigned to every vertex.

In our model, the items are all *distinct 2-grams* seen for a webpage. A 2-gram for a given webpage is a contiguous sequence of 2 words within its HTML representation. The set of 2-grams has been used as a summary of the content of a webpage [14]; the richer a content area is, the more distinct 2-grams. It also corresponds to the classical ROUGE-N [15] measure used in *text summarization*: the higher number of 2-grams a summary shares with its text, the more faithful the summary is. The set of items associated to each vertex plays an important role in the scoring function (see Definition 4), which eventually leads to select a set of webpages for crawling.

A webpage is a well-formed HTML document, and its Document Object Model (DOM) specifies how objects (i.e. texts, links, images, etc.) in a webpage are accessed. Hence, a *root-to-link* path is a location of the link (i.e. <a> HTML tag) in the corresponding DOM tree. Fig. 1 shows a DOM tree representation of a sample webpage and Fig. 2 illustrates its *root-to-link* paths.

Following Definition 1, each edge of the graph is labeled with a *root-to-link* path. Assume there is an edge $e(u,v)$ from vertex u to v, then a label $l(e)$ for edge e is the *root-to-link* path of the hyperlink pointing to v in vertex (i.e. webpage) u. Navigation patterns will thus be (see Definition 3) path expressions over root-to-link paths.

Two webpages reachable from the root of a website with paths p_1 and p_2 whose label is the same are said to be *similar*.

Consider the scoring of a navigation pattern (see Definition 4). We can note the following: the higher the number of requests needed to download pages comprised by a navigation pattern, the lower the score; the higher the number of distinct 2-grams in pages comprised by a navigation pattern, the higher the score.

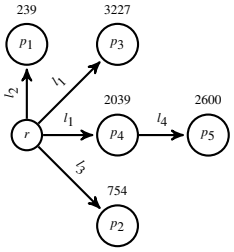

Fig. 4. Before clustering

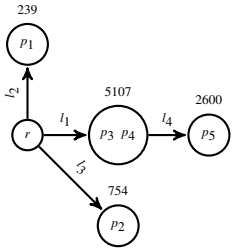

Fig. 5. After clustering

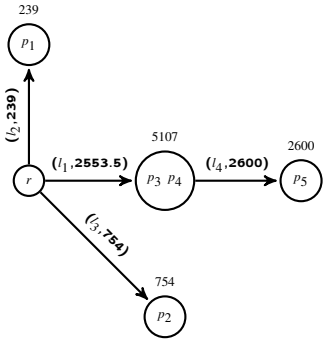

Fig. 6. After scoring

4 Deriving the Crawling Strategy

Simple example. Consider the homepage of a typical web forum, say `http://forums.digitalspy.co.uk/`, as the entry point of the website to crawl. This webpage may be seen as a two different regions. There is a region with headers, menus and templates, that are presented across several webpages, and is considered as a non-interesting region from the perspective of archiving the main content of the website. The other region at the center of the webpage is a content-rich area which should be archived. Since pages are generated by a CMS (vBulletin here), the underlying templates have a consistent structure across similar webpages. Therefore the links contained in those pages obey regular formating rules. In our example website, the links leading to blog posts and the messages within an individual post have some layout and presentational similarities.

Fig. 1 presents a simplified version of the DOM tree of the example entry point webpage and its *root-to-link* paths are shown in Fig. 2. Fig. 4 shows a truncated version of the generated graph for the corresponding site. Each vertex represents a unique webpage in the graph. These vertices are connected through directed edges, labeled with *root-to-link* paths. Each vertex of the graph is assigned a number of distinct 2-grams seen for the linked webpage (e.g. 3,227 distinct 2-grams seen for p_3). Furthermore, the set of webpages (i.e. vertices) that share the same path (i.e. edge label) are clustered together (see Fig. 5). The newly clustered vertices are assigned a collective 2-gram set seen for all clustered webpages. For instance, the clustered vertex $\{p_3, p_4\}$ has now 5,107 distinct 2-gram items. After clustering, all possible navigation patterns are generated for the graph. This process is performed by traversing the directed graph. Table 3 exhibits all possible navigation patterns. Afterwards, each navigation pattern (a combination of *root-to-link* paths) is assigned a score. The system does not compute the score of any navigation pattern that does not lead the crawler from the entry point to an existing webpage. Once all possible navigation patterns are scored then the navigation pattern with highest score is selected (since the highest score ensures the archiving of the core contents). Here, the navigation pattern l_1 is selected. The process of assigning the score to the navigation patterns keeps going after each selection for navigation patterns not selected so far. Importantly, 2-gram items for already selected vertices are not considered again for non-selected navigation patterns. Therefore, in the next iteration, the navigation

Input: entry point r, dynamic sitemap d, navigation pattern expansion depth $expDepth$, navigation-step k, a set of attributes a, completion ratio cr
Output: a set of selected navigation patterns SNP
$siteMap \leftarrow generateSiteMap(r,d)$;
$clusteredGraph \leftarrow performClustering(siteMap)$;
$navigationPatterns \leftarrow getNavigationPatterns(r, clusteredGraph, k, expDepth, a)$;
$NP \leftarrow updateNavigationPatterns(navigationPatterns)$;
while *not cr* **do**
  $topNP \leftarrow getTopNavigationPattern(NP, SNP)$;
  $SNP \leftarrow addToSelectedNP(topNP)$;
  $NP \leftarrow removeSubNavigationPatterns(topNP)$;

Algorithm 1: Selection of the navigation patterns

pattern $l_1 l_4$ does not consider items from webpages retrieved by the l_1 navigation pattern. The process of scoring and selecting ends when no interesting navigation pattern is left.

Detailed description. ACEBot mainly consists of two phases: learning and intensive crawling. The aim of the learning phase is to first construct the sitemap and cluster the vertices that share a similar edge label. A set of crawling actions (i.e. best navigation patterns) are learned to guide massive crawling in the intensive crawling phase.

Algorithm 1 gives a high-level view of the navigation pattern selection mechanism for a given entry point (i.e. the home page). Algorithm 1 has six parameters. The entry point r is the home page of a given website. The Boolean value of the parameter d specifies whether the sitemap of the website should be constructed dynamically. The argument k defines the depth (i.e. level or steps) of navigation patterns to explore. The Boolean *expDepth* specifies whether to limit the expansion depth of navigation patterns to a fixed value of 3; this is typically used in webpages with "Next" links. The argument a passes the set of attributes (e.g. id and class) that should be considered when constructing navigation patterns. cr sets the completion ratio: the selection of navigation patterns ends when this criterion is met.

The goal of the learning phase is to obtain useful knowledge for a given website based on a few sample pages. The sitemap construction is the foundation of the whole crawling process. The quality of sampled pages is important to decide whether learned navigation patterns target the content-rich part of a website. We have implemented a double-ended queue (similar to the one used in [12]), and then fetched the webpages randomly from the front or end. We have limited the number of sampled pages to 3,000, and detailed experiments (see Section 5) show that the sample restriction was enough to construct the sitemap of any considered website. The *generateSiteMap* procedure takes a given entry point as parameter and returns a sitemap (i.e. site model).

The procedure *performClustering* in Algorithm 1 clusters the vertices with similar edge labels. It performs breadth-first traversal over the graph, starting from each root till the last destination vertex. For instance, in Fig. 5, vertex p_3 and p_4 share the label l_1 and thus are clustered together. The 2-gram measure is also computed for each clustered vertex. More precisely, similar nodes are clustered when cluster destination vertices share an edge label, such as a list of blog posts where the label l_2 is shared across vertices. Assume vertex v' has an incoming edge from vertex v with label l_1, and also vertex v' has an outgoing edge to vertex v'' with similar label l_1. Since v' and v'' share an edge label,

these vertices will be clustered. For instance, page-flipping links (e.g. post messages that may exist across several pages) usually have the same *root-to-link* path. These types of navigation patterns end with a + (for example /html/body/div[contains(@class, "navigation")])+), that indicates the crawling action should be performed more than once on similar webpages during intensive crawling.

Once the graph is clustered, *getNavigationPatterns* extracts all possible navigation patterns for each root vertex $r \in R$. The procedure takes three parameters *clusteredGraph*, r, and k as input. The procedure generates the navigation patterns using a depth-first traversal approach where depth is limited to k (i.e. number of navigation-steps). node. Hence, a set of navigation patterns are generated, starting from the root vertex (i.e. the navigation patterns that do not start with the root vertex are ignored) and counting the k number of navigation-steps. This step will be performed for each root vertex and *updateNavigationPatterns* will update the set of navigation patterns *NP* accordingly.

The *getTopNavigationPattern* procedure returns a top navigation pattern on each iteration. The procedure takes two parameters *NP* (a set of navigation patterns), and *SNP* (a set of selected navigation patterns) as input. This procedure applies the subset scoring function (see Definition 4) and computes the score for each navigation pattern. $items(NP)$ is computed by counting the total number of distinct 2-grams words seen for all vertices that share the navigation pattern *NP*. The size of the navigation pattern *NP* (i.e. $size(NP)$) is the total number of vertices that shares the *NP*. The *SNP* parameter is passed to the procedure to ensure that only new data rich areas are identified. More precisely, assume the l_1l_2 navigation pattern is already selected. Now the scoring function for navigation pattern $l_1l_2l_3$ does not take into account the score for navigation pattern l_1l_2, but only the l_3 score will play a role in its selection. Eventually, it guarantees that the system always selects the navigation patterns with newly discovered webpages with valuable content. The *removeSubNavigationPatterns* procedure removes all the sub navigation patterns. For instance, if navigation pattern $l_1l_4l_5$ is newly selected, and there already exists the navigation pattern l_1l_4 in *SNP*, then l_1l_4 will be removed from *SNP*.

The selection of navigation patterns ends when all navigation patterns from the set *NP* are selected or when some other criterion is satisfied (e.g. *completion ratio cr* condition reached). Then, the system will launch the intensive crawling phase and feed the selected navigation patterns to the crawler for massive crawling.

5 Experiments

In this section, we present the experimental results of our proposed system. We compare the performance of ACEBot with AAH [14] (our previous work, that relies on a hand-written description of given CMSs), iRobot [12] (an intelligent crawling system for web forums), and GNU wget[4] (a traditional crawler), in terms of efficiency and effectiveness. Though wget is relatively simple software as far as crawlers are concerned, we stress that any other traditional crawler (e.g. Heritrix) will perform in the same way: as no structure analysis of the website is executed, the website will be exhaustively downloaded.

[4] http://www.gnu.org/software/wget/

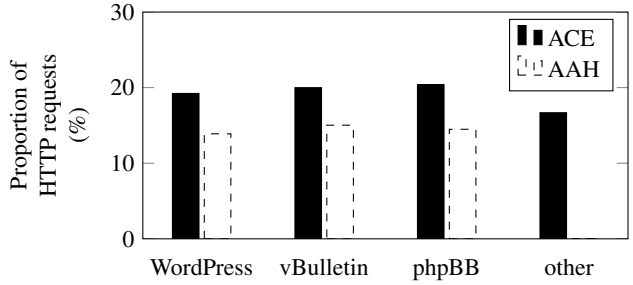

Fig. 7. Total number of HTTP requests (excluding overhead), in proportion to the total size of the dataset

Experimental setup. To evaluate the performance of ACEBot at web scale, we have carried out the evaluation of our system in various settings. We first describe the dataset and performance metrics and different settings of our proposed algorithm. We have selected 50 websites from different application domains (totaling nearly 2 million webpages) with diverse characteristics, to analyze the behavior of our system for small websites as well as for web-scale extraction with both wget (for a full, exhaustive crawl), and our proposed system. To compare the performance of ACEBot with AAH, 10 websites (nearly 0.5 million webpages) were crawled with both ACEBot and AAH (note that AAH only works on selected CMS which prevents a comparison on the larger dataset).

In the learning phase, the site map of a given website is constructed either from the whole mirrored website or from a smaller collection of randomly selected sample pages, as detailed previously. We found that ACEBot requires a sample of 3,000 pages to achieve optimal crawling quality on large websites, comparable to what was done for iRobot [12] (1,000 pages) and a supervised structure driven crawler [10] (2,000 pages).

We consider several settings for our proposed Algorithm 1. The additional parameters d, cr, k, and a form several variants of our technique: The sitemap d may be dynamic (limiting to 3,000 webpages; default if not otherwise specified) or complete (whole website mirror). The completion ration cr may take values 85%, 90%, 95% (default). The level depth k is set to either 2, 3 (default), or 4. The attributes used, a, will be set to *id*.

We have compared the performance of ACEBot with AAH and GNU wget by evaluating the number of HTTP requests made by these crawlers vs the number of useful content retrieved. We have considered the same performance metrics used by AAH [14], where the evaluation of number of HTTP requests is performed by simply counting the requests. In the case of ACEBot, we distinguish between the number of HTTP requests made during the intensive downloading phase (or that would have made during this phase but were already done during that phase) and the *overhead* of requests made during the learning phase for content not relevant in the massive downloading phase (which is bounded by the sample size, i.e., 3,000 in general). Coverage of useful content is evaluated by the proportion of 2-grams in the crawl result of three systems, as well as by the proportion of external links (links to external websites) retrieved.

Crawl efficiency. We have computed the number of pages crawled with ACEBot, AAH, and GNU wget, to compare the crawl efficiency of the three systems (see Fig. 7). Here, wget (or any other crawler not aware of the structure of the website) obviously crawls 100% of the dataset. ACEBot makes 5 times fewer requests than a blind crawl, and slightly more than AAH, the latter being only usable for the three CMS it handles. The numbers in Fig. 7 do not include overhead, but we measured the overhead to be 8% of the number of useful requests made by ACEBot on a diversified sample of websites, which does not significantly change the result: this is because the considered websites were generally quite large (on average 40,000 pages per site, to compare with the sample size of 3,000, which means that on the total dataset the overhead cannot exceed 7.5%).

The results shown in Fig. 8 plot the number of seen 2-grams and HTTP requests made for a selected number of navigation patterns. The numbers of HTTP requests and discovered 2-*grams* for a navigation pattern impact its score, and thus its selection. Therefore a navigation patterns with one single page, but with many new 2-*grams* may be selected ahead of a navigation pattern with many HTTP requests. Fig. 8 elaborates that prospect, where the 10th selected navigation pattern crawls a large number of pages but this navigation pattern was selected only because of a higher completion ratio.

Crawl effectiveness. ACEBot crawling results in terms of coverage of useful content are summarized in Fig. 1 and 9. The coverage of useful content and external links for different navigation steps (levels) is shown in Fig. 1. Limiting the navigation patterns to level $k = 2$ or 3 results in fewer HTTP requests, and a performance of 96% content with $cr = 95\%$ completion ratio. However, level 3 performs better across many websites in terms of effectiveness, as important contents exist at link depth 3. Once the learned navigation patterns achieve the 95% coverage of 2-*grams* vs whole blind crawl, they will be stored in a knowledge base for future re-crawling. The proportion of external link coverage by ACEBot is also given in Table 1. Since ACEBot selects the best navigation patterns and achieves higher content coverage, over 99% of external links are present in the content crawled by ACEBot for the whole dataset.

Fig. 9 depicts the performance of ACEBot for different completion ratios for 10 selected websites, each with 50,000 webpages. The selection of navigation patterns ends when the completion ratio has been achieved. The experiments have shown that a higher (and stable) proportion of *2-grams* is seen with a completion ratio of over 80%.

Comparison to AAH. The experiments of AAH [14] are performed for 100 websites. To compare ACEBot to AAH more globally, we have crawled 10 of the same websites (nearly 0.5 million webpages) used in AAH [14]. ACEBot is fully automatic, whereas the AAH is a semi-automatic approach (still domain dependent) and thus requires a hand-crafted knowledge base to initiate a bulk downloading of known web applications. Over 96 percent crawl effectiveness in terms of *2-grams*, and over 99 percent in terms of external links is achieved for ACEBot, as compared to over 99 percent content completeness (in terms of both *2-grams* and external links) for AAH. The lower content retrieval for ACEBot than for AAH is naturally explained by the 95% target completion ration considered for ACEBot. However the performance of AAH relies on the hand written crawling strategy described in the knowledge base by a crawl engineer. The crawl engineer must be aware of the website structure for the crawled website, to effectively

Table 1. Performance of ACEBot for different levels with dynamic sitemap for the whole data set

Level	Requests	Content (%)	External Links (%)	Completion ratio (%)
	376632	95.7	98.6	85
2	377147	95.8	98.7	90
	394235	96.0	99.1	95
	418654	96.3	99.2	85
3	431572	96.6	99.3	90
	458547	96.8	99.3	95
	491568	96.9	99.4	85
4	532358	97.1	99.4	90
	588512	97.2	99.4	95

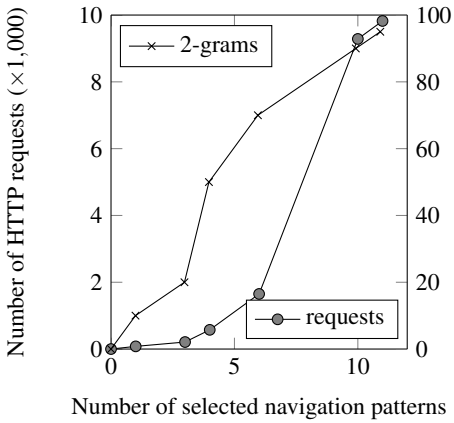

Fig. 8. HTTP requests and proportion of seen 2-grams for 10 websites

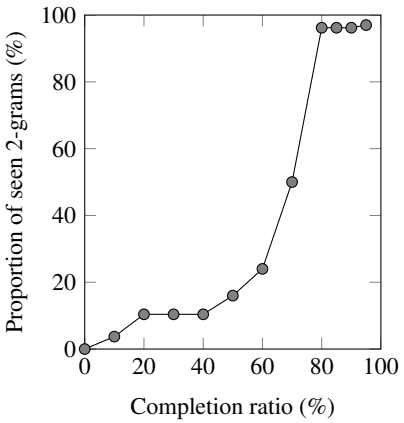

Fig. 9. Proportion of seen 2-*grams* for different completion ratios for 10 websites

download the important portion, as contrasted to our fully automatic approach, where one does not need to know such information for effective downloading and the crawler automatically learns the important portions of the website. The current approach makes 5 times fewer HTTP requests as compared to 7 times for AAH (See Fig. 7).

Comparison to iRobot. We have performed the comparison of our approach with the iRobot system [12]. iRobot is not available for testing because of intellectual property reasons. The experiments of [12] are performed just for 50,000 webpages, over 10 different forum websites (to compare with our evaluation, on 2.0 million webpages, over 50 different websites). To compare ACEBot to iRobot, we have crawled the same web forum used in [12]: `http://www.tripadvisor.com/ForumHome` (over 50,000 webpages). The completeness of content of the our system is nearly 97 percent in terms of *2-grams*, and 100 percent in terms of external links coverage; iRobot has a coverage of *valuable content* (as evaluated by a human being) of 93 percent on the same website. The number of HTTP requests for iRobot is claimed in [12] to be 1.73 times

less than a regular web crawler; on the `http://www.tripadvisor.com/ForumHome` web application, ACEBot makes 5 times fewer requests than wget does.

6 Conclusions

We have introduced an Adaptive Crawler Bot for data Extraction (ACEBot), that relies on the inner structure of webpages, rather than on their content or on URL-based clustering techniques, to determine which pages are important to crawl. Extensive experiments over a large dataset have shown that our proposed system performs well for websites that are data-intensive and, at the same time, present regular structure.

Our approach is useful for large sites, for which only a small part (say, 3,000 pages) will be crawled during the learning phase. Further work could investigate automatic adjustment of the number of pages crawled during learning to the size of the website.

References

1. Gibson, D., Punera, K., Tomkins, A.: The volume and evolution of web page templates. In: WWW. (2005)
2. Q-Success: Usage of content management systems for websites. `http://w3techs.com/technologies/overview/content_management/all` (2015)
3. Alpert, J., Hajaj, N.: We knew the web was big... `http://googleblog.blogspot.co.uk/2008/07/we-knew-web-was-big.html` (2008)
4. Faheem, M.: Intelligent Content Acquisition in Web Archiving. PhD thesis, Télécom ParisTech (2014)
5. Liu, Z., Ng, W.K., Lim, E.P.: An automated algorithm for extracting Website skeleton. In: DASFAA. (2004)
6. Kao, H.Y., Lin, S.H., Ho, J.M., Chen, M.S.: Mining web informative structures and contents based on entropy analysis. IEEE Trans. Knowl. Data Eng. (2004)
7. Crescenzi, V., Merialdo, P., Missier, P.: Fine-grain web site structure discovery. In: WIDM. (2003)
8. Crescenzi, V., Merialdo, P., Missier, P.: Clustering web pages based on their structure. Data Knowl. Eng. **54**(3) (2005)
9. Bertoli, C., Crescenzi, V., Merialdo, P.: Crawling programs for wrapper-based applications. In: IRI. (2008)
10. Vidal, M.L.A., da Silva, A.S., de Moura, E.S., Cavalcanti, J.M.B.: Structure-driven crawler generation by example. In: SIGIR. (2006)
11. Jiang, J., Song, X., Yu, N., Lin, C.Y.: Focus: Learning to crawl web forums. IEEE Trans. Knowl. Data Eng. (2013)
12. Cai, R., Yang, J.M., Lai, W., Wang, Y., Zhang, L.: iRobot: An intelligent crawler for web forums. In: WWW. (2008)
13. Blanco, L., Dalvi, N.N., Machanavajjhala, A.: Highly efficient algorithms for structural clustering of large websites. In: WWW. (2011)
14. Faheem, M., Senellart, P.: Intelligent and adaptive crawling of web applications for web archiving. In: Proc. ICWE. (2013)
15. Lin, C.Y., Hovy, E.: Automatic evaluation of summaries using n-gram co-occurrence statistics. In: HLT-NAACL. (2003)