Crawling the Deep Net
Final Term Paper

Munaf Arshad, Catherine Eng, Maeda Hanafi

August 2016

Key terms— Surface Web; Deep Web; Web Crawler;

1 Introduction

The normal web, or the "*surface web*" is everything part of the World Wide Web indexed and accessed directly by us, using such search engines. A web crawler is used to crawl through the list of available URLs and catalogue each page. A database is constructed and an entry for fast access of that page at a later date based on content, some specific keys or meta-tags on the page is made. Each time a new page is added, it is inspected by these web-spiders and is added to a list of reachable pages.

The *deep net* is a term coined by Mike Bergman to refer to Internet content "not indexed by standard search engines". [1] This content may be banking records, private databases or even our social networking messages. Deep web also contains dynamic web pages, blocked sites, unlinked sites, private sites and all non HTML/Scripted content we encounter. It is estimated that surface web is approximately only 20TB of data compared to deep web which is as large as 7.5 Petabytes! [2] The large amount and the vast variety of types of data in the deep web are some of the challenges researchers face when crawling through the deep web.

The motivation behind indexing the deep net lies in the amount of data it holds; estimates have measured "several orders of magnitude larger than than the surface web." [1] Moreover, majority of deep web's data is *structured*, data that is tabulated, e.g. databases (the opposite being *unstructured* data such as YouTube comments). Hence, the problem of dealing with deep net data is not in restructuring that data, but rather accessing it.

In other words, the majority of research challenges in deep net revolve around automatically accessing and querying its data. Several algorithms have been proposed to tackle with the different kinds of data that the deep net holds as well as how to gain access that data, whether it be from computer vulnerabilities to automatic, simple HTML form querying. In this paper, we will discuss the two major things:

1. the kinds of data hidden in the deep net, and
2. the techniques that researchers have come up with to tackle the problem.

2 Algorithms

Several methods have been explored to crawl the deep web, ranging from the early efforts that combined human-intervention and semi-automation [20, 18, 6, 8] to fully automated end-to-end

crawlers [17, 11] targeting various kinds of resources, e.g. multi-form, textual-based forms, etc. Early attempts at crawling the web were limited to specific sites and required human annotators to provide values to certain forms. Hence, crawling the web entailed the following problems:

1. Choosing the specific database to search (resources of a given domain) - *The database selection problem*, also referred to as site locating. The goal of crawlers is collect as much information across distributed resources, and the database selection problem refers to selecting a small set of resources or databases to send a query at minimum cost.

2. Searching the chosen database. Since no static URL is provided to the target data, the crawler must query the data and understand the interface of that data. Two methods exist to understand the interface and query the data: 1) *schema matching* [15, 23, 22] and 2) *probing* [16, 8, 9, 13]. In schema matching, a crawler learns the semantics of the data's query interface. Schema matching essentially maps values to concepts of a given form. Another aspect of searching a given database includes probing the underlying data. The goal of probing is to form an accurate description of the underlying data via sampling queries to its interface.

Initially, early works [20, 18, 6, 8, 12] assumed the resource or database was given. Hence, such works were limited to a particular site. Eventually, works have extended over data across sites. Works in searching the chosen database extend into multiple aspects and majority fall in either schema matching or probing. The major challenge with schema matching was its creation and maintenance. One advantage of probing is its ability to crawl text-based query interfaces, e.g. search query interfaces.

In this paper, we will cover selected algorithms that targeted these problems, compare and contrast their advantages and weaknesses. We will begin with the early works and then finally describe the most advanced of crawlers that attempts to solve both of the challenges: choosing the relevant databases and searching it.

2.1 Human-Intervened Crawling with Schema Matching

Of the earliest works on web crawling, Hidden Web Exposer (HiWE) [20] stood out due to the work's focus on crawling the hidden web (or in today's terms deep web) that takes advantage of information extracted from the HTML forms, refilling a schema matching data structure to facilitate future crawls. The work essentially revolves around learning query interfaces with the use of schema matching.

HiWE operates on the following constraints: 1) *task specificity* and 2) initial, human-provided form values. Task specificity refers to the domain on which the crawler should search data on, e.g. data such as papers, publications, articles on the semi-conductor industry. The task specificity is stored in a database called the *task specificity database*, which holds the information required to ensure the crawler queries data relevant to the task. Specifically, this database is stored in a schema match data structure modeled as fuzzy sets called a *Label Value Set* (LVS). An LVS stores labels, also referred to as concepts e.g. company, and its corresponding values, e.g. Microsoft, Apple, etc. Since it is a fuzzy set, each associated value is assigned some weight indicating the confidence of the crawler of the association. Although HiWE requires initial values for forms, it gathers other possible values as it crawls and it updates the LVS.

Given the task specificity and the initial values, HiWe works as follows:

1. Converts the form into an *Internal Form Representation* using a heuristic approach for webpage layout analysis where visually close forms inputs and text descriptions (labels) are associated with each other,

2. Matches a given form and LVS with a set of value assignments used as input to that form,

3. Analyzes the response of the form and uses the information to tune future queries to forms and to update the LVS.

The form representation makes a distinction between two types of form input: *finite domains* and *infinite domains*. Finite domains have a set of values already embedded in the page, e.g. selection lists. Infinite domains are free-form inputs, e.g. text boxes that take as input strings. The representation refers to the names or descriptions of the form inputs as *labels*.

The matching of values to the form labels is done with a match function. If a form input is a finite domain, the HiWE simply tries all possible values. If a form input is a infinite domain, the match function tries to match the input's label to the labels in the LVS, e.g. a form label of "Enter state" will best match the label "State" from the LVS. HiWE employs string matching algorithms to find label matches. To maximize efficiency, HiWE's matching function also assigns a value representing the crawler's confidence of a label's match to a set of values. Such values are required to be at least greater than some threshold. The threshold is based on the minimum of some aggregation function, and the paper experimented with three possible aggregation functions: 1) fuzzy conjunction: the threshold is the minimum of value assignment amongst the constituent values, 2) average: the threshold is the minimum is the average of the constituent values, 3) probabilistic: the threshold depends on how likely that the choice of a value is useful, e.g. if many of the individual values are very low then this results in a match having a lower rank. By calculating threshold for values, HiWE prunes low quality value assignments and is able to improve submission efficiency. In their experiments, they found the probabilistic threshold to be too aggressive; it pruned valid form submissions.

Once the matches are ranked, HiWE sends them as input to the query interface and analyzes the forms. During analysis, HiWE adds values to the LVS in order to gather more information for the next crawl. LVS tables can be filled via wrapping an external data source, e.g. Yahoo directory, etc, manual entry, or taking advantage of the finite domain form elements, e.g. selection forms, since by default they contain labels and its associated values.

HiWE provides important contribution to early web crawling efforts by introducing form analysis and an operational model of a hidden web crawler. However, it is severely limited in that it requires a human in its operation and requires specification of the resources and domain, e.g. "Given a database of science articles, find papers on X topic." Moreover, HiWE is limited only to forms with more than 3 inputs, and doesn't perform well with short forms, since short forms lack more descriptive labels needed for HiWE to correctly convert into an internal form representation. Another limitation is in the schema match structure; managing such structure eventually becomes hard to maintain. Later on, in the following sections, we describe how other methods have circumvent these limitations.

2.2 Probing

One of the most significant challenges is the difference between availability of surface web and deep web documents. Documents in the surface web are typically stored as document files whereas in the deep web, data is often dynamic and computed on demand (e.g., keyword query entered into a web interface accessing a database). Additionally, sites can restrict web crawling through configuration

(e.g., the robots.txt file). This means that traditional web crawling techniques to index documents no longer prove effective when searching in the deep web. Probing is currently implemented in two forms: with human interaction and with automation.

2.2.1 Probing with Human Interaction

Scripts can be created to retrieve data from sites and developed either manually or semi-automatically using wrapper generators. While scripts are effective and efficient, they can require large amounts of human interaction since they need to be tailored to a given Web site and search interface, thereby limiting the adaptability of such a solution. [8]

An alternative option is to use a hidden web crawler, which provides meaningful attribute assignment of the forms encountered while the page is crawled automatically. Scanning through the form elements exposes potential input values which can be submitted by the crawler. However, for open ended text fields, knowledge of the domain is needed and the set of potential values must be sent to the crawler. This again, necessitates significant human interaction and performance depends heavily on the quality of the input data. Additionally, forms with fewer attributes may be ignored (e.g., Stanford's HiWE crawler) and this method will primarily derive its results from using the form's multi-attribute fields as input. [8]

2.2.2 Probing with Automation

Many attempts have been made to automate probe querying to locate data in the deep net. One suggested proposal by Barbosa and Freire [8] utilizes a two part sampling based algorithm which discovers keywords to achieve the algorithm's best coverage:

1. First the SampleKeyword algorithm (Algorithm 1) selects a word from the initial form page and submits a query. If an error is returned, choose the next keyword and submit the query until a valid keyword is found. Next, check if a stopword (e.g., "the", "a", "for") is used. If a stopword is used, then generate the candidate keyword with the stopword, otherwise just the candidate keyword is sufficient. Now, while the number of submitted queries is less than max submission count, submit a query and use the results to continue to add new keywords or update the frequency of the existing candidate keywords.

2. Next, part two of the algorithm, ConstructQuery (Algorithm 2), is called. This function uses a "greedy strategy" to construct the highest coverage. The keyword with the highest frequency from the candidate keyword list is selected and submitted as a query. Even though query construction may be expensive, once a high coverage query is identified, it can be reused for future searches (e.g., such as when a query refreshes indexes periodically).

The team put the algorithm through a practical test of varying conditions: a) choosing sites with known and varied collection sizes, b) different tunings of query parameters to avoid overwhelming sites with long running inefficient queries (e.g., max terms to 10), c) use of stopwords considered (showed that if omitted, keywords generated are more relevant, d) sites indexed by numbers or characters), e) number of iterations. At condition e), most sites showed candidate keyword sets converging after five iterations, at fifteen iterations the candidate keyword set showed 90% coverage with exception of two sites; one of the anomalous sites took fifty iterations to achieve 79.8% coverage, this is most likely due it's collection size which is multiple factors larger than the other sites.

The team also tested other properties, such as the a) collection anomalies due to fields set as required/optional, b) fields accepting stopwords inconsistenly across a site, c) keyword selection,

e.g. keyword extracted using either the title or description with the description, which they found to have better coverage, and d) filtering extraneous details from the results (such as ads, navigation bar which lead to only slightly better coverage which is expected since the pages are more content rich with more keywords leading to better coverage).

Overall, the Barbosa and Freire algorithm's effectiveness is impacted by the inputs used to the algorithm (e.g., number of iterations) as well as site properties (e.g., size of the site, indexing properties, data homogeneity). This work essentially focuses on building multi-keyword queries that can return a large part of a given document collection. It is also one of the first noted automated query probing algorithms available. [8]

On the other hand, another query probing system available is QProber created by Ipeirotis, Gravano, and Sahami [13]. QProber automates the classification of searchable text databases by adaptively searching the database based on document classifiers without retrieving any documents. The goal of their document classifier is not to classify documents but rather extract *classification rules*, logical rules that association a conjunction of words to some topic. The rules form query probes and their algorithm analyzes the results of those queries (specifically the number of matches associated with a category) without having to retrieve actual documents. Their method was evaluated over 130 real web databases of varying sizes, and the technique produced very accurate database classification results. To classify an entire deep net database, less than 200 queries of four or less words are used. Additionally their approach is effective even for databases without "hidden" content as long as a web search interface is available.

Additionally, the algorithm has the side effect of helping "uncooperative databases", which are databases that cannot strongly associate with a topic. Using the algorithm, incomplete databases can classify their documents and match to existing topics created from other databases which are able to produce complete content summaries. [16]

QProber's algorithm is constructed in four parts [13]:

1. First the document classifier (tool used to automatically categorize documents into common topics such as "Computers", "Health" and later on used to generate rules) is trained with a set of "pre-classified documents" (development set). This step removes popular (e.g., "am", "so", "the") and unpopular words; determined by Zipf's law [4] of feature selection. Next an information-theoretic feature selection algorithm eliminates terms with least significance in order to determine document class distribution. This piece of the algorithm decreases the number of features in a structured way so a smaller list of words create the classifier. Once the features are selected (e.g., words) a machine learning algorithm is used to create the document classifier. Common document classifiers such as RIPPER [10], C4.5[19], Bayes, and Support Vector Machines (SVM) were used in the team's experiment.

2. Second, the classification rules are gathered from the document classifier and query probes are generated. The classification rules are as follows:

   - Pick a random word from a dictionary and a one word query is sent to the database
   - Retrieve top-N documents returned by the query database
   - Extract words from each document and update the list and frequency of each word
   - Classify the documents in the collection

   For example Figure 1 shows, the seeded top categories (e.g., "Arts", "Computers", "Health") and rules then place keywords within a category (e.g., "Perl AND Java -> Programming"). Most of the documents are expected to be retrieved into the same top level category for this

site (e.g., "Computers"). Instead of retrieving all the documents, the number of matches reported for this query (using the results page with the line "X documents found") and use this number to measure the document database match to this rule. From this number, a good estimate of the Coverage (total number matches for query probe) and Specificity (percentage of documents in this category) vectors in a database D are determined. Then the number of documents D in a category C can be found based on the number of query probe results using the category rules.

The algorithm works by generating rules iteratively. At each iteration, the rules are developed of different lengths, with a different number of terms in the antecedents. At the first iteration, only rules with one term are used. If the term weight is higher than a threshold, this term qualifies to form a rule, since the availability of this term is sufficient to allow classification of the document into the category. Rules are formed without negations and may include conjunctions only. Once the rules have been created with one term, the algorithm moves to the next iteration which creates rules with two terms, and continues till all rules are created.

3. Queries are adaptively issued by extracting and adjusting match count with a confusion matrix. Since documents classifiers may mistakenly classify documents into incorrect categories or leave documents unclassified (if no rules match), QProber introduces an algorithm to adjust the initial probing results. The algorithm generates query probes from the rules and probes the database of unseen pre-classified documents, next creates an additional confusion matrix and set x equal to the sum of the number of matches from the document's category. Next, the columns are normalized by dividing column with the number of documents in the development set. The result is a normalized confusion matrix. The researchers multiply this confusion matrix with the Coverage vector to yield the correct number of documents for each category in the development set.

4. Finally, databases are classified according to adjusted query match counts. Each category node is annotated with ECoverage (e.g., total matches in a category query probe) and ESpecificy (e.g., percentage of documents in the parent category as well as in the category itself). Researchers set thresholds on both fields (e.g., Especificity=0.5 and Ecoverage=100). To check the classification, if both thresholds are within limits, then the category can be explored to classify the document otherwise discarded as irrelevant to the classification.

To test the practicality of the algorithm, the team evaluated several factors including different thresholds, controlled database set, multiple document classifiers, and flat versus hierarchical algorithms for comparison. [13, 16] However, there is still future work to be done. For example, if the match count is not returned then the document database rule match (Part 2 of the algorithm) cannot be calculated corrected, or when the results are bounded by an upper limit with only a portion of results returned (though database truncations found are relatively small), or if malicious databases deliberately returned incorrect match counts or results, or if automatic querying is restricted.

This particular approach by Ipeirotis, Gravano and Sahami presents an exciting solution and creates a very elegant way to approach the query probing problem. The rules are dynamically modified and updated as more information is received, which makes it flexible and adaptable to any domain. Additionally, the algorithms consistently produce precise classifications and efficient performance as verified by the research team.

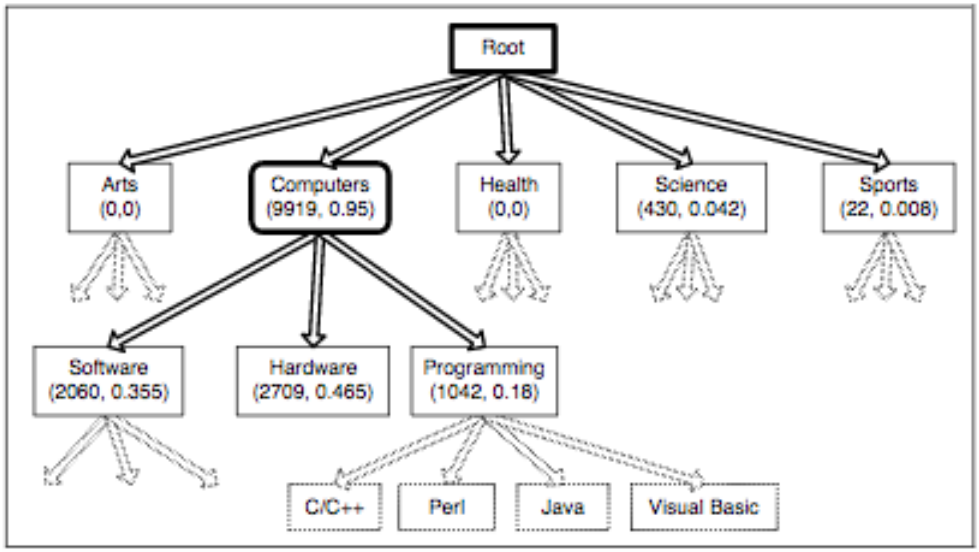

Figure 1: QProber Classification of ACM Digital Library Scored by ECoverage, ESpecificity

2.3 Site Location and In-site Exploration with a Two-Stage Crawler for Efficiently Harvesting Deep-Web Interfaces

In this section, we describe a two-stage web crawler which aims to peruse the dynamic data in the deep web [3] to produce high quality, efficient and relevant results to search topics called SmartCrawler [24]. It improves upon the short comings of both generic crawlers (that fetch all searchable forms and cannot focus on a specific topic) [21] and focused crawlers (that can be led to pages without targeted forms) [5].

SmartCrawler [24] works in two stages: Site locating and in-site exploring. Figure 2 shows a block diagram for the entire process.

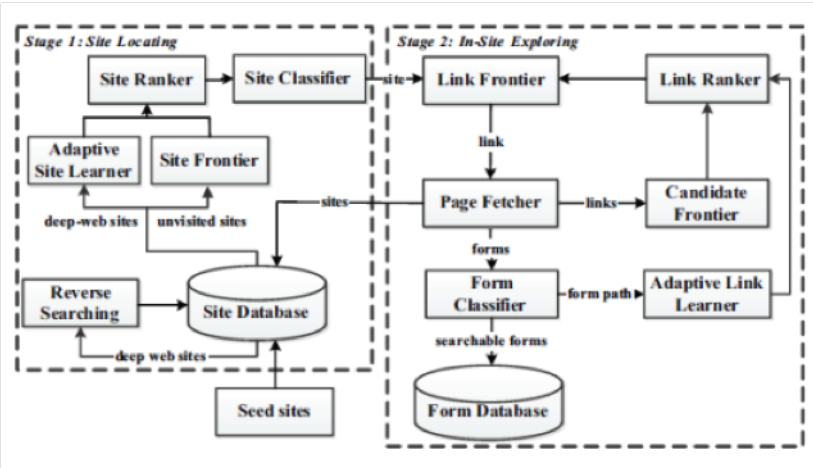

Figure 2: The two stage architecture of SmartCrawler [24]

Seed sites are sites that are given to SmartCrawler as candidates. The crawler follows URLs in these seed sites to look at other domains and pages. When a certain no. of URLs have been visited

and a small number is left, reverse searching is triggered. This is searching of the known deep web sites for center pages (highly ranked pages with links to other domains). These are then fed into the site database to keep it populated. The Site Frontier then fetches homepage URLs from the site database to be ranked by the Site Ranker. An adaptive site learner continuously extracts features from each site and keeps on learning so that each time a site is ranked according to relevancy; it's priority is assigned as accurately as possible. The Site Classifier categorizes pages as relevant or irrelevant according to the content presented on their home pages.

Once the most relevant site has been found (set highest in priority), the program proceeds with stage two. Here an efficient in-site exploration for excavating searchable forms takes place. The link frontier stores the page's links while page fetcher fetches all pages that need to be examined. The examination is done by the forms classifier while links are again forwarded to the candidate frontier. After the classifier has decided about the form, the forms database is updated and the extracted features are again used by the adaptive link learner to improve the link ranking module: the link ranker.

The Site locating stage consists of three important steps for any given topic: Site Collection, Site Ranking and Site Classification. A typical crawler would visit all and any links found on a webpage when crawling but the smartcrawler, in contrast, minimizes the number of already visited URLs and maximizes the number of deep websites. One problem is that looking for external links in the page wouldn't necessarily produce a sufficient list for the Site Frontier. This is why this algorithm proposes two crawling strategies: Reverse Searching and two-level site prioritizing.

Reverse Searching randomly picks a known deep website or a seed site and uses a search engine to look for center and other relevant pages. The result from the search engine is parsed to extract links and these links' pages are downloaded. If the pages contain any related searchable forms, or the number of seed sites in the page is larger than a threshold, the sites are declared relevant and are forwarded to the site frontier.

Incremental site prioritizing helps the whole process to be resumable and reusable with better results over time. Past crawling and other prior information is initialized into both the rankers (site and link), and as the smartcrawler follows out of site links, the unvisited sites are saved into one of two queues in the site frontier. Sites which are classified as relevant by the Site Classifier and the Form Classifier are kept in the high priority queue. Sites that are only judged as relevant by the Site Classifier are kept in the low priority queue. Each queue has a score for each element as well and the low priority one is used as extra sites when the high one has been exhausted. Following

are the algorithms for both these strategies.

Algorithm 1: Reverse searching for more sites

```
   input  : seed sites and harvested deep websites
   output: relevant sites
 1 while  # of candidate sites less than a threshold do
   |  /* pick a deep website                                          */
 2 |  Site = getDeepWebsite(siteDB, seedSites)
 3 |  resultPage = reverseSearch(Site)
 4 |  links = extractLinks(resultPage)
 5 |  for link in links do
 6 |  |  page = downloadPage(link)
 7 |  |  relevant = classify(page)
 8 |  |  if relevant then
 9 |  |  |  relevantSites = extractUnvisitedSite(page)
10 |  |  |  Output relevantSites
11 |  end
12 end
```

Algorithm 2: Incremental Site Prioritizing

```
   input  : siteFrontier
   output: searchable forms and out-of-site links
 1 HQueue=SiteFrontier.CreateQueue(HighPriority)
 2 LQueue=SiteFrontier.CreateQueue(LowPriority)
 3 while siteFrontier is not empty do
 4 |  if HQueue is empty then
 5 |  |  HQueue.addAll(LQueue)
 6 |  |  LQueue.clear()
 7 |  site = HQueue.poll()
 8 |  relevant = classifySite(site)
 9 |  if relevant then
10 |  |  performInSiteExploring(site) Output forms and OutOfSiteLinks
   |  |   siteRanker.rank(OutOfSiteLinks)
11 |  |  if forms is not empty then
12 |  |  |  HQueue.add (OutOfSiteLinks)
13 |  |  else
14 |  |  |  LQueue.add(OutOfSiteLinks)
15 |  |  end
16 end
```

Once the site frontier has enough sites, selecting the most relevant one for crawling is an issue. This is why the Site Ranker assigns a score to each website.

The Classifier is a Naïve Bayes one which uses a feature vector constructed from the stemming and stop words from the homepage content of the site. This is how topical relevance comes into play to improve the system overtime.

Once a site has been declared relevant, it is searched for forms. The goal is to do this quickly and to cover as much web directory of the site as possible. These goals are met using two crawling strategies: Stop-Early and Balanced link prioritizing. The stop early performs a breadth-first search to achieve broader coverage (as 72% interfaces and 94% of webdata can be found within the depth

of 3 [14]). However, as we set the stopping criteria from, either the max crawling depth, max crawling pages in each depth, predefined number of forms for each depth or predefined number of pages without forms for each depth etc. this method can miss some important results. Therefore, alongside this balanced link prioritizing strategy is considered. According to it, as some directories would have more 'relevant' names than the others, it would be unfair to prioritize according to directory names. Therefore, a link tree is created with an equal no. of nodes from each directory and they are explored for searchable forms.

When done, the link ranker prioritizes and assigns scores to links from candidate frontier. A high score would be assigned to links that would be most similar to links that point directly for forms. It continues to learn and improve from the adaptive link learner. The form classifier decides whether the form searched is relevant or not and improves the learner.

Scanning deep web interfaces has always been a challenge not only because of the vast size of this part of the world wide web but also because of its dynamic content. This algorithm takes that into consideration and adds adaptive learning modules to its both stages. Because of a lot of these bells and whistles, compared to other generic and focused crawlers (e.g. Form-Focused Crawler (FFC)) we would assume the time and space complexity would have worsened. However even after being more complex, this algorithm has not only increased efficiency of its results compared to other adaptive algorithms such as Adaptive Crawler for Hidden-web Entries (ACHE) [7] but it also has outperformed it in terms of both Running time and Search Results. Figure 3 shows this.

Domain	Running Time —		Searchable Forms	
	ACHE	*SmartCrawler*	ACHE	*SmartCrawler*
Airfare	7h59min	6h59min	1705	3087
Auto	8h11min	6h32min	1453	3536
Book	8h21min	7h32min	599	2838
Job	8h50min	8h8min	1048	4058
Hotel	8h37min	6h54min	2203	4459
Movie	10h9min	6h26min	677	1449
Music	7h59min	6h29min	776	1347
Rental	8h1min	6h38min	1149	2538
Product	7h50min	6h29min	386	571
Apartment	8h7min	6h7min	1844	3958
Route	8h0min	6h36min	1061	3471
People	8h3min	6h43min	232	677

Figure 3: Comparison of ACHE and SmartCrawler [24]

Even after so having much to offer, considering the huge amount of time required to run this algorithm for simple domains/search terms one could argue this is a perfect solution. A lot of this algorithm relies on the performance of its learners and classifiers and a few rogue results could not only deteriorate performance by tenfolds but also waste hours of processing power and time. After analyzing the pros and cons of this algorithm we conclude it is a step in the right direction, however the road ahead is still a long one.

3 Conclusion

In this paper, we reviewed the range of works done in crawling the deep web. We described the challenges in this field, namely database selection and how to search that database. We described two methods to deal with searching the database: schema matching and probing. We also described

systems related to each approach. The systems covered in this paper came with differing capabilities, ranging for human-intervened systems to fully automated systems. As the field develops, one should expect future works to aim beyond full automation and into various other kinds of query interfaces including those that focus efficiency.

References

[1] Deep web. `https://en.wikipedia.org/wiki/Deep_web`. Accessed: 2016-08-01.

[2] Deep web vs. dark web. `https://www.youtube.com/watch?v=rCZyvY6E6Zs`. Accessed: 2016-08-01.

[3] Idc worldwide predictions 2014: Battles for dominance –and survival – on the 3rd platform. `http://www.idc.com/research/Predictions14/index.jsp`. Accessed: 2016-08-01.

[4] Zipf's law. `https://en.wikipedia.org/wiki/Zipf%27s_law`. Accessed: 2016-08-01.

[5] Luciano Barbosa and Juliana Freire. Searching for hidden-web databases. In *WebDB*, pages 1–6, 2005.

[6] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden-web entry points. In *Proceedings of the 16th international conference on World Wide Web*, pages 441–450. ACM, 2007.

[7] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden-web entry points. In *Proceedings of the 16th international conference on World Wide Web*, pages 441–450. ACM, 2007.

[8] Luciano Barbosa and Juliana Freire. Siphoning hidden-web data through keyword-based interfaces. *Journal of Information and Data Management*, 1(1):133, 2010.

[9] Jamie Callan and Margaret Connell. Query-based sampling of text databases. *ACM Transactions on Information Systems (TOIS)*, 19(2):97–130, 2001.

[10] William W Cohen and Yoram Singer. Learning to query the web. In *AAAI Workshop on Internet-Based Information Systems*, pages 16–25, 1996.

[11] Jared Cope, Nick Craswell, and David Hawking. Automated discovery of search interfaces on the web. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 181–189. Australian Computer Society, Inc., 2003.

[12] Hasan Davulcu, Juliana Freire, Michael Kifer, and IV Ramakrishnan. A layered architecture for querying dynamic web content. *ACM SIGMOD Record*, 28(2):491–502, 1999.

[13] Luis Gravano, Panagiotis G Ipeirotis, and Mehran Sahami. Qprober: A system for automatic classification of hidden-web databases. *ACM Transactions on Information Systems (TOIS)*, 21(1):1–41, 2003.

[14] Bin He and Kevin Chen-Chuan Chang. Statistical schema matching across web query interfaces. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 217–228. ACM, 2003.

[15] Bin He and Kevin Chen-Chuan Chang. Automatic complex schema matching across web query interfaces: A correlation mining approach. *ACM Transactions on Database Systems (TODS)*, 31(1):346–395, 2006.

[16] Panagiotis G Ipeirotis and Luis Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 394–405. VLDB Endowment, 2002.

[17] Jayant Madhavan, David Ko, Łucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's deep web crawl. *Proceedings of the VLDB Endowment*, 1(2):1241–1252, 2008.

[18] Alexandros Ntoulas, Petros Zerfos, and Junghoo Cho. Downloading hidden web content. *Technicalreport, UCLA*, 2004.

[19] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[20] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. 2000.

[21] Denis Shestakov. Databases on the web: national web domain survey. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, pages 179–184. ACM, 2011.

[22] Jiying Wang, Ji-Rong Wen, Fred Lochovsky, and Wei-Ying Ma. Instance-based schema matching for web databases by domain-specific query probing. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 408–419. VLDB Endowment, 2004.

[23] Wensheng Wu, Clement Yu, AnHai Doan, and Weiyi Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 95–106. ACM, 2004.

[24] Feng Zhao, Jingyu Zhou, Chang Nie, Heqing Huang, and Hai Jin. Smartcrawler: A two-stage crawler for efficiently harvesting deep-web interfaces. 2015.