Web Archive Profiling Through Fulltext Search

Sawood Alam[1], Michael L. Nelson[1], Herbert Van de Sompel[2], and David S. H. Rosenthal[3]

[1] Computer Science Department, Old Dominion University, Norfolk, VA (USA)
{salam,mln}@cs.odu.edu
[2] Los Alamos National Laboratory, Los Alamos, NM (USA)
herbertv@lanl.gov
[3] Stanford University Libraries, Stanford, CA (USA)
dshr@stanford.edu

Abstract. An archive profile is a high-level summary of a web archive's holdings that can be used for routing Memento queries to the appropriate archives. It can be created by generating summaries from the CDX files (index of web archives) which we explored in an earlier work. However, requiring archives to update their profiles periodically is difficult. Alternative means to discover the holdings of an archive involve sampling based approaches such as fulltext keyword searching to learn the URIs present in the response or looking up for a sample set of URIs and see which of those are present in the archive. It is the the fulltext search based discovery and profiling that is the scope of this paper. We developed the *Random Searcher Model (RSM)* to discover the holdings of an archive by a random search walk. We measured the search cost of discovering certain percentages of the archive holdings for various profiling policies under different RSM configurations. We can make routing decisions of 80% of the requests correctly while maintaining about 0.9 recall by discovering only 10% of the archive holdings and generating a profile that costs less than 1% of the complete knowledge profile.

Keywords: Web Archive, Memento, Archive Profiling, Random Searcher

1 Introduction

The number of public web archives supporting the Memento protocol [17] natively or through proxies continues to grow. Currently, there are 28[1], with more scheduled to support Memento in the near future. The Memento Aggregator [13], the Time Travel Service[2], and other services, need to know which archives to poll when a request for an archived version of a file is received. Efficient Memento routing in aggregators is desired from both the aggregators' and archives' perspective. Aggregators can reduce the average response time, improve overall throughput, and save network bandwidth. Archives benefit by the reduced

[1] http://labs.mementoweb.org/aggregator_config/archivelist.xml
[2] http://timetravel.mementoweb.org/

number of requests for which they have no holdings, hence saving computing resources and bandwidth. In December 2015, soon after the surge of OldWeb.today[3] many archives struggled with the increased traffic. We found that fewer than 5% of the queried URIs are present in any individual archive other than the Internet Archive. In this case, being able to identify a subset of archives that might return good results for a particular request becomes very important.

We envision following four approaches to discover the holdings of an archive:

CDX Profiling – If an archive's CDX (Capture/Crawl inDeX) files[4] are available, then we can generate profiles with complete knowledge of their holdings. In our recent work [2] we established a generic archive profiling framework and explored the *CDX Profiling*. We examined an extended set of 23 different policies to build archive profiles and measured their routing efficiency.

Fulltext Search Profiling – Random query terms are sent to the fulltext search interface of the archive (if present) and from the search response we learn the URIs that it holds. These URIs are then utilized to build archive profiles. In this paper we are exploring the *Fulltext Search Profiling* within the framework established by our recent work. It is important to note that some web archives (including the Internet Archive) do not provide fulltext search, hence this approach is not applicable for them.

Sample URI Profiling – A sample set of URIs are used to query the archive and build the profile from the successful responses. This is quite wasteful, as <5% of the sample URIs are found in any archive. AlSum et al. explored the *Sample URI Profiling*, but only on Top-Level Domains (TLDs) [3]. We want to further explore this approach with many profiling policies.

Response Cache Profiling – This approach depends on the response data collected by an aggregator over a period of time as queries are made to the archive. Cached responses are analyzed to learn about their holdings. As a result the *Response Cache Profiling* is based on what people were looking for as opposed to what is in the archives. This approach is different from the other three in a way that it is a usage-based profiling approach while the other three are content-based.

In this paper we introduce a *Random Searcher Model (RSM)* to randomly explore a collection and discover K distinct documents. Unlike the *Random Surfer Model* [4] which constructs the graph of web pages based on the hyperlinks, RSM assumes that the documents in the collection are connected with co-occurring terms. Randomly picking a term from a document and searching for that term in the collection yields many other documents that share the term. Selecting a document from the returned document set to choose the next random term and repeat this process allows us to know about the holdings of the collection. To generalize the RSM and to accommodate several possibilities we experimented with some variations in how to select the next search term.

[3] `http://oldweb.today/`

[4] `http://archive.org/web/researcher/cdx_file_format.php`

2 Related Work

Query routing is common practice in various fields including meta-searching and search aggregation [10, 15, 16, 7, 9, 8, 19]. Sanderson et al. first explored Memento query routing by exhaustive CDX profiling [14]. They collected CDX files from various IIPC member archives and generated profiles based on the complete URI-Rs (original Resource URIs) from them (we denote it as *URIR Profile* in this paper). This approach gives complete knowledge of the holdings in each participating archive, hence queries can be routed precisely to archives that have any mementos (URI-M) for the given URI-R. It is a resource and time intensive task to generate such profiles and some archives may be unwilling or unable to provide their CDX files. Such profiles are large enough in size (typically, a few billion URI-R keys) that they require special infrastructure to support fast lookup. Acquiring fresh CDX files from various archives and updating these profiles regularly is difficult.

Many web archives tend to limit their crawling and holdings to some specific TLDs, for example, the British Library Web Archive prefers sites with `.uk` TLD. AlSum et al. explored *Sample URI Profiling* based on TLD [3] in which they recorded *URI-R Count* and *URI-M Count* under each TLD for twelve public web archives. Their results show that they were able to retrieve the complete TimeMap [17] in 84% of the cases using only the top 3 archives and in 91% of the cases when using the top 6 archives. This simple approach can significantly reduce the number of queries generated by a Memento aggregator with some loss in coverage.

The two efforts described above have explored extreme cases of profiling. We believe that an intermediate approach that gives flexibility with regards to balancing accuracy and effort can result in better and more effective routing. Our earlier work [2] establishes the general framework of flexible archive profiling. It describes the general model of partial URI, date, language, and composite keys and associated statistical measures. We also introduced a profile serialization format called CDXJ (CDX-JSON) [1] to make it easy to store, disseminate, merge, and perform efficient lookups. In our earlier work we generated profiles from CDX files where we knew the complete holdings of the archive. We then examined costs and routing efficiencies of 23 different profiling polices.

Bornand et al. explored *Response Cache Profiling* and implemented Memento routing by building binary classifiers from the aggregator cache data (in this case the classifier is functionally equivalent to an archive profile) [5]. They report a 77% reduction in the number of requests and a 42% reduction in response time while maintaining a 0.847 recall value.

There have been many efforts on crawling the hidden web that have no hyperlinks and are accessible only by filling out HTML forms [11, 18, 12]. Our keyword search based archived content discovery is related to these efforts because archived contents span over a long period of time, which causes a disconnect between old and contemporary pages. As a result, hyperlink based shallow crawling might only discover a temporal sub-graph of the holdings.

3 Methodology

If a web archive provides fulltext searching in its collection, it can be leveraged to discover the holdings of the archive. In this approach, we send some random query terms to the archive and collect the resulting URIs returned from the archive. Although with each successful response we learn some new URIs, the learning rate slows down as we go forward because of the fact that some of the URIs returned in a response may have been already seen in earlier attempts. This follows Heaps' Law [6].

With carefully chosen values of various advanced search attributes and selection of suitable list of keywords we can affect the parameters of the Heaps' Law and maximize the learning rate. In this section we analyze different fulltext approaches to discover holdings of an archive. We also explore different ways to perform the search when the language of the archive is unknown or a list of suitable keywords is not available for that language.

For our experiments we chose the Archive-It hosted North Carolina State Government Web Site Archive collection[5] which is the largest collection in Archive-It and provides fulltext search. Since we have Archive-It data (up to 2013) hosted in our dark archive at Old Dominion University (ODU), it was easier for us to perform the coverage analysis on this dataset.

We started our experiment by searching for stop-words. For example when we searched for the term "a" it returned 26M+ results, which is very close to the number of the HTML resources in the collection up till March 2013. However, each page only contains 20 results, hence, in order to learn all the 26M+ URIs we will have to make 1.3M+ HTTP GET requests. Our goal was to make as few requests as possible to learn enough diverse set of URIs in the collection. Additionally, this approach cannot be generalized, as not all archives will behave the same on stop-words.

In the next step we built static and dynamic word lists and searched for those words as query terms. For each term we only record the URIs in the first resulting page and move on to the next word in the list. Having a configurable pagination would have benefited us by choosing a larger number of results per page, but Archive-It has it fixed to 20 results and does not allow any changes.

Top Words – We collected top the 2,000 English nouns[6] and used them as the query terms. We accumulated the first page results to plot the learning curve, but also extracted other information such as the result count for each search term. As expected, these top terms yielded high values for the result count for each terms. Additionally, each term yielded more than one page of results, hence no effort was unsuccessful. We also observed that the response time is correlated with the result count, so the same number of top terms would take longer to fetch than a random word list.

[5] `https://www.archive-it.org/collections/194`
[6] `http://worddetail.org/most_common/nouns`

Random Linux Dictionary – We ran the same procedure on a randomly chosen set of 2,000 unique words from the built-in Linux dictionary. This time the average number of results per terms was lower, but there were many terms for which the collection returned no results or fewer than 20 (page size) results.
Dynamically Discovered Word List – The above two experiments were based on the static lists of words. This static word list approach requires the knowledge of the language (and field) of the collection to choose a static list of words for that language, which may not be easily available. Additionally, the list would be finite and may not be enough to discover a sufficient number of the collection holdings. To overcome this issue, we build a model (discussed in Section 4) that can dynamically discover new words from the searched pages and utilize those words to perform further searching. We introduced different policies in the dynamic discovery model based on how the new words are learned and how the next word for searching is chosen. We then analyzed the learning rate and the number of HTTP requests for each policy.

4 Random Searcher Model

To perform searches on a static or dynamic word list, we developed a general *Random Searcher Model (RSM)* that can be configured to operate in one of the four modes with the help of configurable *Vocabulary Seeding* and *Word Selection* policies. To understand the model we will discuss different data structures, policies, operation modes, and procedures separately then put them together to describe the overall working.

4.1 Data Structure

The RSM has the following data structures to hold various intermediate statistics and states:

Vocabulary – A data structure to hold the list of words to be searched. Depending on some policies it may or may not allow duplicates. This data structure should provide the number of total or unique words in the list. The data structure should support functions to overwrite the list, add more words to the list, pop a random word from the list, and remove all occurrences of the popped word in the list.
SearchLog – A data structure to hold the record of each search attempt. It contains the searched word, attempt result (success or failure), and any additional meta information such as the response result count and newly discovered URIs. An implementation may choose to offload some results to a separate data structure or include more attributes for analysis. This data structure should provide the number of total or successful searches. The data structure should support functions to add new records, querying if a given word is present, and randomly selecting a successful searched word.

ResultBank – A dictionary like data structure to hold all the discovered URI-Rs and their respective memento counts. This data structure should provide the number of total URI-Rs (or dictionary keys). The data structure should support functions to add new records and iterate over all the records.

4.2 Policies

Policies control the behavior of the RSM. We have defined two different policies with various valid configuration values as described below:

Vocabulary Seeding Policy – This policy controls how the *Vocabulary* is seeded and how often. Valid values are:

- *Static* – The *Vocabulary* is manually seeded in the beginning with a static list of search keywords. It allows seeding only once and the procedure terminates when all the seeded keywords are consumed.
- *Progressive* – The *Vocabulary* is initialized with a few words, but it is aggressively overwritten after each successful search with the newly discovered words, except when there are no new words discovered.
- *Conservative* – The *Vocabulary* is initialized with a few words and new words discovery is only performed when all the words from the *Vocabulary* are consumed. The *Vocabulary* is only reseeded when it is empty.

Word Selection Policy – This policy controls how search words are selected from the *Vocabulary*. Valid values are:

- *Popularity Biased* – Randomly select one word from the *Vocabulary* where the probability of a word being selected is proportional to the term frequency in the *Vocabulary* normalized by the total number of terms in the *Vocabulary*. A simple implementation of this policy would consider the *Vocabulary* as a bag of random words (with duplicates allowed) to select a random word from it. There are other memory efficient approaches possible, but this simple approach illustrates the concept naturally.
- *Equal Opportunity* – Randomly select one word from the list of unique words in the *Vocabulary*.

4.3 Operation Modes

An operation mode is a valid combination of *Vocabulary Seeding* and *Word Selection* policies (as mapped in Table 1). Not all combinations of the two policies are valid. We recognize the following four as valid modes of the *RSM*:

Static – This mode operates on a static word list that is known in advance so it does not have to make additional fetches to discover more words. However, finding a suitable word list for a collection could be difficult.
PopularityBiased – In this mode the searcher randomly picks a URI from the returned result and fetches that page to discover new words. Once the words are

Table 1: RSM Operation Mode Mapping with Policies

Operation Mode	Vocabulary Seeding	Word Selection
Static	Static	Equal Opportunity
PopularityBiased	Progressive	Popularity Biased
EqualOpportunity	Progressive	Equal Opportunity
Conservative	Conservative	Equal Opportunity

fetched, it randomly picks a word from that and repeats the search operation. In this way the searcher has to fetch a page after every search attempt to search for the next word. Selection of the words is random, but the duplicates are not removed so the words with higher frequency in the page have higher chance of being selected.

EqualOpportunity – This mode works the same way as the *PopularityBiased* mode does, but it picks the next word from the unique list of words so a rare word on the page has the same probability of being selected as a high frequency one.

Conservative – The *PopularityBiased* and the *EqualOpportunity* modes have the drawback of being twice as costly in terms of number of HTTP requests as compared to a static word list of the same size. The reason for this is that after every successful search, there is a page fetch to learn new words. However, this *Conservative* mode does not throw away previously learned words and consumes each of them. It makes a page fetch to learn new words only when all the previously learned words are consumed. This reduces the number of HTTP requests significantly.

4.4 Procedures

The RSM has the following public procedures:

Initialize() – This method initializes a Random Searcher instance with supplied configuration options and *Vocabulary* seed.

TerminationCondition() – This method returns True if the conditions are met to terminate the *NextWord* iterator. It returns False otherwise.

NextWord() – This method pops the next word from the *Vocabulary* based on the Word Selection Policy (and removes all the occurrences of the word in the *Vocabulary*, if any). If the *Vocabulary* is empty and the terminating condition is not met then it randomly picks a successful searched word from the *SearchLog*. This is a generator function that can act as an iterator until the configured termination condition is met.

Search() – This method searches the collection for a given search keyword and populates the *SearchLog* and the *ResultBank* with appropriate values from the search result. Depending on the configured policies it may also select a URI from the result to extract words from it to seed the *Vocabulary*.

ExtractWords() – This method fetches the page at the given URI, sanitizes it (by stripping off the markup, scripts, and styles), tokenizes the text to split in

non-empty words, and returns the bag of words (duplicates included, if any) in the order they appeared in the document.
GenerateProfile() – This method iterates over all the items in the *ResultBank* and generates an *Archive Profile* based on the supplied profiling policy.

To run the *Random Searcher* an instance of the RSM is initialized with one of the four possible modes, some initial seed words, termination condition configuration, tokenizer pattern, and other configurations. Then the *NextWord* generator is iterated over to discover next word for searching until it hits the termination condition. For every word, the *Search* method is called which internally performs the collection lookup and updates various data structures of the RSM instance. Once the iterator terminates, *GenerateProfile* method is called to serialize collected statistics in an *Archive Profile*.

5 Implementation

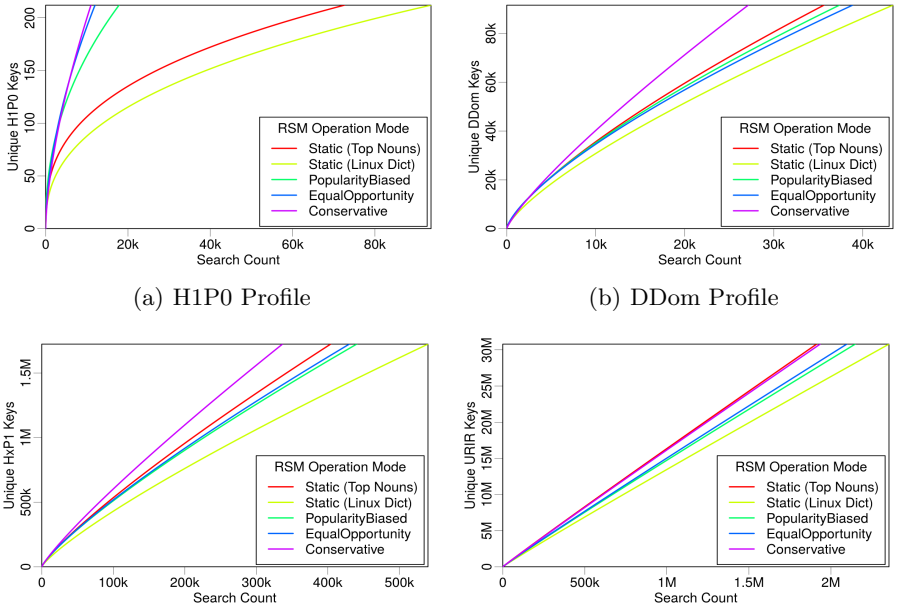
We implemented the RSM in Python and made the code available[7]. However, due to the lack of a uniform search API across archives, the implementation is not generic enough to run against any archive. The page scraping part of the code that extracts useful pieces of information from the search result page and assembles them in a data structure needs custom implementation for each archive. The Archive-It search interface has an undocumented JSON response that we used to avoid HTML parsing. Our implementation has some additional intermediate data structures and logging in place for the sake of analysis which is not needed for production purposes.

6 Evaluation

To evaluate the RSM we estimated the search cost of different operation modes for discovering certain portion of the archive holdings in terms of a given profiling policy. To evaluate generated archive profiles we measured how much of the archive holdings we must discover in order to gain a satisfactory Recall and corresponding routing efficiency for a given profiling policy. In this analysis we have used four different profiling policies. Examples are derived from the URI `https://www.news.BBC.co.uk/Images/Logo.png?width=80&height=40`.

H1P0 – Only TLDs are used as keys (212 unique keys in the collection). E.g., `uk)/`.
DDom – Only registered domain name is used as keys (91,629 unique keys in the collection). E.g., `uk,co,bbc)/`.
HxP1 – All host segments and one path segment are used as keys (1,724,284 unique keys in the collection). E.g., `uk,co,bbc,news)/Images`.
URIR – SURTed URI-Rs are used as keys (30,800,406 unique keys in the collection). E.g., `uk,co,bbc,news)/Images/Logo.png?height=80&width=200`.

[7] `https://github.com/oduwsdl/archive_profiler`



(a) H1P0 Profile (b) DDom Profile

(c) HxP1 Profile (d) URIR Profile

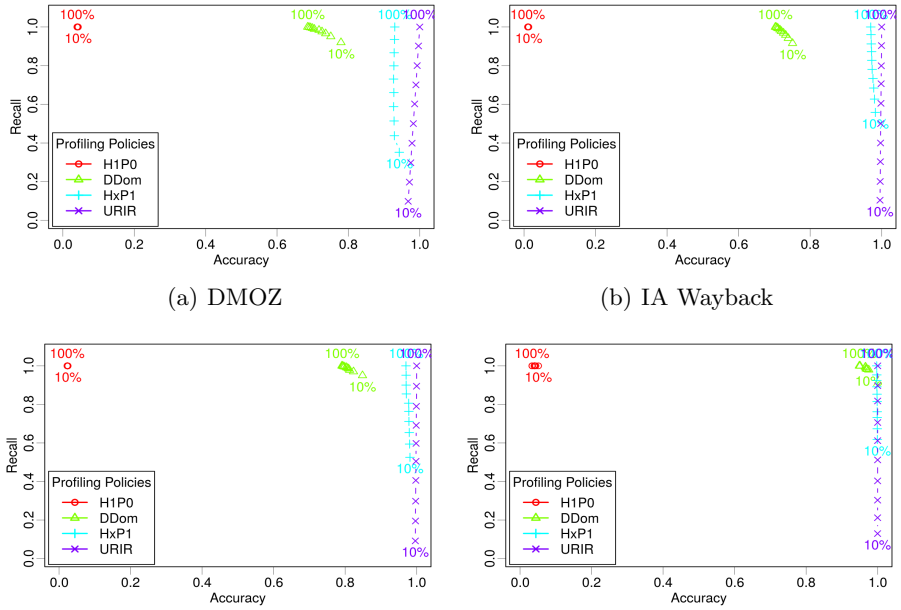
Fig. 1: Searches Needed vs. Required Coverage

6.1 Estimating Searches Needed

Figure 1 illustrates the projected estimate of the search cost for each of the four profiling policies based on the initial 2,000 searches. Each graph is extended up to the 100% limit of each profiling policy on the Y-axis. For example, there are total 212 unique TLDs in the collection, hence the upper Y-axis limit in Figure 1(a) is set to 212. The RSM operation modes in which additional HTTP request is made to fetch a Memento to extract words cause additional HTTP GET overhead. Table 2 shows the HTTP cost of each RSM operation mode with respect to the corresponding query cost. The value of δ is quite small as compared to C. In our experiments we found that for $C = 2,000$ the value of δ was 8. Which means for making 2,000 queries we only needed eight additional HTTP GETs to extract new words in *Conservative* mode. The *Conservative* mode is an overall winner. It shows a higher learning rate and does not require a static list of words to be supplied. It is also good because it does not have much overhead HTTP request cost for the term discovery.

Table 2: Cost Comparison of RSM Operating Modes

Operation Mode	**Query Cost**	**HTTP Cost**
Static	C	C
PopularityBiased	C	$2 * C$
EqualOpportunity	C	$2 * C$
Conservative	C	$C + \delta$ (where $\delta \ll C$)



(a) DMOZ (b) IA Wayback

(c) Memento Proxy (d) UKWA Wayback

Fig. 2: Incremental Accuracy vs. Recall as a Function of Archive Knowledge

6.2 Profile Routing Efficiency

Unlike CDX analysis, a fulltext search based archive profile will often be created based on partial knowledge of the archive holdings. This may cause incomplete recall in Memento routing, which means that an archive might have some Mementos unknown to the profile and as a result it may not have a matching key in it to route the query there. Hence, it is important to analyze the incremental changes in the confusion matrix as we know more and more of the archive's holdings. Table 3 illustrates the confusion matrix in the Memento routing context. For this analysis we selected the early ten years of UKWA dataset as the gold dataset. Then we extracted the unique URI-Rs from it and randomized it. The randomized URI-R list was then split into ten equal chunks. We then profiled these chunks incrementally using different policies to see how these policies perform against a sample set of query URIs when we know only 10%, 20%, 30%... to 100% of the archive. We repeated this process for four different query URI sample sets of one million each. In each case we calculated the confusion matrix and plotted Accuracy vs. Recall graph to estimate the routing efficiency. Accuracy here means how often an archive profile is correct in routing or not routing a query to the archive.

Figure 2 shows that if the complete archive is known, we should choose higher order profiles such as HxP1. The HxP1 profile has the Accuracy almost as good as the URIR profile. From our earlier work we know that the cost of the HxP1 profile is less than one sixth of the cost of the URIR profile. Additionally, the

Table 3: Confusion Matrix of Memento Routing

Predicted \ Actual	**Present in the Archive**	**Not in the Archive**
Routed to the Archive	True Positives (TP)	False Positives (FP)
Not Routed to the Archive	False Negatives (FN)	True Negatives (TN)

HxP1 profile has higher Recall than the URIR profile when the complete archive is not known. However, since we cannot afford to lose much of the recall, we favor smaller profiles such as DDom when we have partial access to the archive (such as when using fulltext search). The DDom profile (that has less than 1% cost as compared to the URIR profile) can have about 0.9 Recall while correctly routing (or not routing) more than 80% of the URIs by only knowing 10% of the archive. Cases where only a tiny fraction of the archive is known by sampling (such as when neither CDX is available nor fulltext search), we favor the smallest profile H1P0/TLD-only to maintain an acceptable Recall value. Even though the URIR profile always yields 100% Accuracy, it suffers from poor Recall. With the complete CDX accessible, the URIR policy costs a lot compared to other policies. Additionally, the URIR policy does not have any predictive powers for unseen URIs, hence maintaining the freshness of the profile is challenging as the archives acquire more URI-Rs.

7 Future Work and Conclusions

So far we only worked on profiles based on URIs. Going forward we want to examine other dimensions such as the Memento date and the content language and a combination of more than one of these. We want to examine how these profiles generated for various archives can be digested into a system that ranks the archives based on the probability of the availability of each queried URI in various archives.

In this paper we examined the *Fulltext Search Profiling*. We developed a *Random Searcher Model* to discover the holdings of archives that support fulltext search. We evaluated the query and HTTP costs to learn certain percentage of the holdings of an archive using RSM under different profiling policies. We also evaluated the routing efficiency in terms of *Accuracy* and *Recall*. We concluded that the DDom profile (that has less than 1% cost as compared to the URIR profile) can have about 0.9 Recall while correctly routing (or not routing) more than 80% of the URIs by only knowing 10% of the archive. Finally, we open-sourced our RSM implementation code.

Acknowledgements

This work is supported in part by the International Internet Preservation Consortium (IIPC).

References

1. Alam, S., Kreymer, I., Nelson, M.L.: Object Resource Stream (ORS) and CDX-JSON (CDXJ) Draft. `https://github.com/oduwsdl/ORS` (2015)
2. Alam, S., Nelson, M.L., Van de Sompel, H., Balakireva, L., Shankar, H., Rosenthal, D.S.H.: Web Archive Profiling Through CDX Summarization. In: Proceedings of 19th International Conference on Theory and Practice of Digital Libraries, TPDL 2015. pp. 3–14
3. AlSum, A., Weigle, M.C., Nelson, M.L., Van de Sompel, H.: Profiling Web Archive Coverage for Top-Level Domain and Content Language. International Journal on Digital Libraries 14(3-4), 149–166 (2014)
4. Blum, A., Chan, T.H., Rwebangira, M.R.: A Random-Surfer Web-Graph Model. In: Proceedings of the Meeting on Analytic Algorithmics and Combinatorics. pp. 238–246. ANALCO '06, Society for Industrial and Applied Mathematics (2006)
5. Bornand, N., Balakireva, L., Van de Sompel, H.: Routing Memento Requests Using Binary Classifiers. In: Proceedings of the 16th ACM/IEEE-CS Joint Conference on Digital Libraries. JCDL '16 (2016)
6. Egghe, L.: Untangling Herdan's law and Heaps' law: Mathematical and informetric arguments. Journal of the American Society for Information Science and Technology 58(5), 702–709 (2007)
7. Gravano, L., Chang, C.C.K., García-Molina, H., Paepcke, A.: STARTS: Stanford Proposal for Internet Meta-Searching. SIGMOD Rec. 26(2), 207–218 (Jun 1997)
8. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying Heterogeneous Information Sources Using Source Descriptions (1996)
9. Liu, L.: Query Routing in Large-scale Digital Library Systems. In: Proceedings of 15th International Conference on Data Engineering. pp. 154–163 (1999)
10. Meng, W., Yu, C., Liu, K.L.: Building Efficient and Effective Metasearch Engines. ACM Computing Surveys (CSUR) 34(1), 48–89 (2002)
11. Ntoulas, A., Zerfos, P., Cho, J.: Downloading Textual Hidden Web Content Through Keyword Queries. In: Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries. pp. 100–109. JCDL '05 (2005)
12. Raghavan, S., Garcia-Molina, H.: Crawling the Hidden Web. Tech. Rep. 2000-36, Stanford InfoLab (2000), `http://ilpubs.stanford.edu:8090/456/`
13. Sanderson, R.: Global Web Archive Integration with Memento. In: Proceedings of the 12th ACM/IEEE-CS Joint Conference on Digital Libraries. pp. 379–380. ACM (2012)
14. Sanderson, R., Van de Sompel, H., Nelson, M.L.: IIPC Memento Aggregator Experiment. `http://www.netpreserve.org/sites/default/files/resources/Sanderson.pdf` (2012)
15. Sugiura, A., Etzioni, O.: Query Routing for Web Search Engines: Architecture and Experiments. Computer Networks 33(1), 417–429 (2000)
16. Tran, T., Zhang, L.: Keyword Query Routing. Knowledge and Data Engineering, IEEE Transactions on 26(2), 363–375 (2014)
17. Van de Sompel, H., Nelson, M.L., Sanderson, R.: HTTP Framework for Time-Based Access to Resource States – Memento. RFC 7089 (2013)
18. Wu, P., Wen, J.R., Liu, H., Ma, W.Y.: Query Selection Techniques for Efficient Crawling of Structured Web Sources. In: Proceedings of the 22nd International Conference on Data Engineering. pp. 47–47. ICDE '06 (2006)
19. Xu, J., Callan, J.: Effective retrieval with distributed collections. In: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval. pp. 112–120. ACM (1998)