Compressed Collections for Simulated Crawling

Alessio Orlandi*
Università di Pisa, Italy
aorlandi@di.unipi.it

Sebastiano Vigna
Università degli Studi di Milano, Italy
vigna@dsi.unimi.it

Abstract

Collections are a fundamental tool for reproducible evaluation of information retrieval techniques. We describe a new method for distributing the document lengths and term counts (a.k.a. within-document frequencies) of a web snapshot in a highly compressed and nonetheless quickly accessible form. Our main application is reproducibility of the behaviour of focused crawlers: by coupling our collection with the corresponding web graph compressed with WebGraph [3] we make it possible to apply text-based machine learning tools to the collection, while keeping the data set footprint small. We describe a collection based on a crawl of 100 Mpages of the `.uk` domain, publicly available in bundle with a Java open-source implementation of our techniques.

1 Introduction

Focused crawling is a term originally given by Chakrabarti *et al.* [6] to denote a crawling activity that gathers *relevant* pages (as opposed to all pages). The notion of relevance is dependent on the particular application: for instance, a user might provide a set of interesting pages as an example.

The main issue in designing a focused crawling policy is the prioritisation of the queue containing the frontier. Since the pages in the frontier are known, but not crawled, to maximise the *harvest rate* (the number of relevant pages gathered averaged over time) it is essential to choose from the frontier either relevant pages, or non-relevant pages that will quickly lead to relevant ones.

Performances of crawling policies can be studied either on the real Internet or via visit simulation on stored graphs. The latter provide a scientifically reproducible playground suitable for comparative studies among strategies and for parameter testing. In our case, simulation requires both the graph structure and the page content. This work has been motivated by the absence of a large standardised reference collection for focused crawling, and more in general for the application of text-feature based machine learning techniques to the web. Such a collection should be:

- *Large*: representing a sizeable portion of the World Wide Web.

*Partially supported by Italian MIUR Project “MAINSTREAM”.

- *Heterogeneous*: containing different topics and types of sites. For instance, a crawl containing pages only from a few domain names is not.
- *Open*: we require the collection to be small enough for distribution via network or optical discs and to be available for other researchers.
- *Easy*: provided with a portable software architecture for fast decompression and access.
- *Shrinkable*: since we expect the collection to be large, we want to include the possibility of sub-collections of different smaller size for incremental testing.

To accommodate the above goals and avoid copyright problems, we assume that page file format and structure are irrelevant toward classification; more precisely, we will restrict the data derived from document content to the *counts* (i.e., the number of occurrences, a.k.a. within-document frequencies) of the terms appearing in the document, and to the document lengths in words. This information, strongly related to the representation of documents in the vector model, is sufficient to rebuild all weighting schemes we are aware of (TF/IDF, BM25, etc.) and it is powerful enough to support many known machine learning tools (e.g. SVMs or Bayesian classifiers). On the other hand, it is clear that our choice has significant limitations, as it discards pieces of information that some focused crawlers could need (e.g. anchor text delimiters or positional information).

There are two issues in replicating a web crawl: describing the graph structure, and describing the page content. For the graph structure, we rely on the WebGraph framework [3]. We will provide instead tools to access quickly the information about the page contents.

2 Archives of Summaries

We now present our main contribution—the design and implementation of what we call a *bitstream archive of summaries*. More precisely, a document summary is a tuple

$$\langle \ell, s, L = \langle t_0, c_0\rangle, \langle t_1, c_1\rangle, \dots , \langle t_{s-1}, c_{s-1}\rangle\rangle,$$

where ℓ is the document length in words, s (the *size* of the summary) is the number of distinct terms appearing in the document and L is a list of s term/count pairs. We assume that our documents are numbered so that the node number in the graph matches its document identifier (albeit content might be missing for some pages due to HTTP server errors).

We want to use *gap encoding* techniques that are common in the compressed storage of inverted indices in this setting. To this purpose, we make the key observation that *terms should be renumbered by frequency rank*. More explicitly, term 0 is the term of highest frequency, and so on.[1] As a result, the term lists contain numbers which are often smaller than the original ones: this phenomenon improves significantly the results of gap encoding.

For coding counts we use the following observation, supported by empirical evidence: terms that have a high frequency (e.g., stopwords) have usually also a high count. Thus, we list counts

[1]The same idea has been used in [1] to store compactly the result of disjunctive queries.

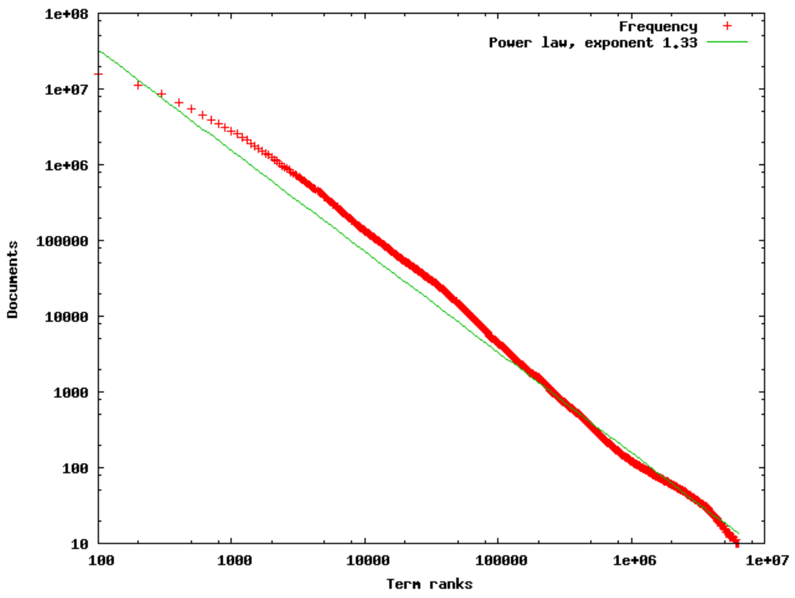

Figure 1: Logarithmic plot of the distribution of renumbered terms, computed on a sample of 11 million documents with a dictionary of over 6 million terms.

in *inverse* order (e.g., starting from the less frequent term) and gap code them. We have of course no guarantee that the counts will be increasing, so we use the mapping $\nu : \mathbf{Z} \to \mathbf{N}$ thus defined:

$$\nu(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ 2|x| - 1 & \text{if } x < 0. \end{cases}$$

The difference between consecutive counts (which can be negative) is mapped through ν and the resulting (small) integer coded using an instantaneous code.

Summaries in the archive are stored contiguously in the format described above.

Choosing the codes. Once the strategy for storing our data is laid out, we just have to discuss which instantaneous codes we should use. As often happens with text, several of the distributions involved are power laws or similar distributions: in Figure 1 we plot the frequency of terms sorted by rank in a sample web collection, and note that, as expected, the distribution is close to a power law. Since the distribution of the difference between two power-law random variables is not known, we again resort to empirical evidence: Figure 2 shows the distribution of gaps, and while the distribution is clearly noisy, it suggests again a power law of slightly larger exponent.

Thus, beside the classical γ and δ codes, we also experimented with ζ codes [4], which were designed for power laws. Our suggestion is δ coding for document lengths, ζ_3 coding for sizes of summaries, γ coding for gaps of term counts, and ζ_2 coding for gaps of term numbers.

Random access. The bit stream containing all documents clearly provides sequential access to the entire collection, as each summary is self-delimiting. On the other hand, simulating a crawler requires random access to the entire archive. To avoid keeping in memory a very expensive array of pointers explicitly, we resort to *rank/select* data structures. Given a set of integers S, ranking an integer x gives the number of elements of S smaller than x, whereas selecting a rank r gives

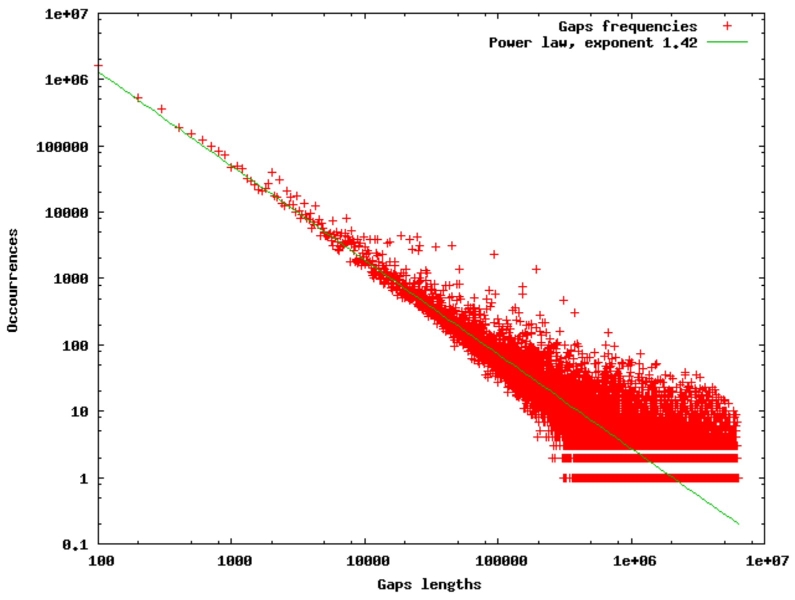

Figure 2: Logarithmic plot of the distribution of gaps of renumbered term for the sample of Figure 1.

the r-th element in S (counting from 0). In particular, we use a broadword implementation [9] of the Elias–Fano representation of monotone sequences [7, 8], which provide (almost) constant-time rank and select operations occupying a space close to the informational-theoretical lower bound.

To obtain efficient random access, we sort the summaries in increasing document identifier order and build two sets. The first set contains the identifiers of pages that are present in the graph but missing from the archive. By ranking in this set we can, given a document identifier d, find its ordinal position in the bit stream (by subtracting the number of missing documents that precede it). Since missing document are a small fraction, this set occupies little space.

The second set contains the bit pointers to each summary. In our sample case (described in the next section), for instance, the Elias–Fano representation provides direct access using less than 13 bits per summary. Subarchives can be easily layered over real archives using just additional rank/select structures.

Of course, summaries contain now data based on a renumbered version of the terms, so we decorate our archive with an auxiliary file containing the permutation inverting the frequency-rank order. As a result, once a summary is read we can map the term indices to their original values. In our implementation this process is transparent to the user.

An open-source Java implementation of the techniques described is publicly available at the Archive4J project web site (`http://archive4j.dsi.unimi.it/`). Bitstream archives can be built easily from MG4J[2] document collections: building an archive involves a preprocessing phase that gathers term statistics and makes it possible to filter terms (e.g., *hapax legomena*). We also provide tools to build archives in a distributed fashion: both the preprocessing and the scanning phases are run locally, and their results can be then combined. The instantaneous codes used for compression can be customised to suit the needs of collections with significantly different term

[2]MG4J is a full-text search engine; see `http://mg4j.dsi.unimi.it/`.

distributions.

3 A Sample Collection

As a sample of the techniques we described, we present a snapshot of about 100 million pages obtained from an original generic crawl of the `.uk` domain performed by the Laboratory of Web Algorithmics in May 2007 using UbiCrawler [2] (crawls made by UbiCrawler have already been used to build collections for evaluation [5]; this particular crawl is being used to build the successor to [5]).

The crawler started from a seed of about 190000 different URLs. For each host, the crawler downloaded up to 50 Kpages with maximum depth 8. Only pages containing text were stored. The crawl was stopped at about 100 Mpages, resulting in about 500 GiB of data stored in WARC/0.9 gzipped format (see [10]).

There are of course a number of choices that have to be made to go from a snapshot to a stream of summaries. We parse HTML pages and apply the Porter stemmer; then, we remove English stopwords and all terms appearing in less than 20 documents. These choices are of course fairly arbitrary, and dictated by the need of eliminating *hapax legomena* and typos, but they are just parameters of our tools, and they can be set differently (actually, we plan to release several versions of the collection with different tradeoffs between space occupancy and accuracy). We remark, moreover, that standardising the preprocessing phase has also the effect of making the classification process further reproducible, by unifying the text preprocessing phase that most algorithms perform.

The bitstream archive storing the resulting summaries occupies about 20 GiB for data. Satellite data that must be loaded into main memory include 150 MiB for pointer data and 35 MiB for the term permutation and frequencies. With the above parameters the archive dictionary size results of about 6 million unique terms. This level of compression, combined with a 1 GiB graph file, makes it possible to use the collection to simulate a crawl of 100 million pages on a standard PC in few hours.

References

[1] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. *Proc. of the 29th annual international ACM SIGIR conference*, pp. 364–371. ACM Press, 2006.

[2] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience* 34(8):711–726, 2004.

[3] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. *Proc. of the 13th Intl. World Wide Web Conference*, pp. 595–601, 2004.

[4] P. Boldi and S. Vigna. Codes for the world wide web. *Internet mathematics* 2(4):405–427, 2005.

[5] C. Castillo, D. Donato, L. Becchetti, P. Boldi, S. Leonardi, M. Santini, and S. Vigna. A reference collection for web spam. *ACM Sigir Forum* 40(2):11–24, 2006.

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

[7] P. Elias. Efficient storage and retrieval by content and address of static files. *J. Assoc. Comput. Mach.* 21(2):246–260, 1974.

[8] R. M. Fano. On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass., n.d., 1971.

[9] S. Vigna. Broadword implementation of rank/select queries. *Proc. of the 7th International Workshop on Experimental Algorithms*, pp. 154–168. LNCS 5038, Springer Verlag, 2008.

[10] WARC file format, ISO/DIS 28500, 2007.