WEB GRAPH COMPRESSION IN MARKOVPR 1.1

L.A. BREYER

Abstract. These notes describe the results of some investigations into reducing the memory requirements of the *MarkovPR 1.0* software for storing, and simulating from, web graphs. The statistics are based upon the Google Programming Contest dataset. Many of the techniques described here have been implemented in the newer version, *MarkovPR 1.1*.

1. Introduction

These notes represent the results of various investigations into reducing the memory requirements of the *MarkovPR 1.0* web graph construction software. The first version of this software, which is described in more detail in (Breyer, 2002), does not fill memory optimally.

With the Google dataset, which consists of 916,428 web documents, *MarkovPR's 1.0* ripper program requires approximately 200 Mb to represent the web graph, and allow page ranking simulations. This includes space to store bidirectional links and some other information associated with each node or web document. We shall refer to these requirements as the *baseline*. In this note we explain how the baseline requirements can be reduced to about 85 Mb on the same dataset. Table 1 summarizes the improved storage requirements, as already implemented in *MarkovPR 1.1*.

The topic of web graph compression and layout witin RAM has previously been considered in Adler and Mitzenmacher (2001), Broder et al. (2000), Suel and Yuan (2001). There are two statistics of main interest for web graph storage: the number of bytes needed for each stored URL, and the number of bytes needed for each outlink. Suel and Yuan report 6.5 bytes per URL, and 1.7 bytes per link. Bharat

Date: August, 2002.

1991 *Mathematics Subject Classification.* Primary 60J, Secondary 60F.

Key words and phrases. Search engines, web graph compression, information retrieval, PageRank, Markov chains.

Average number of bytes per URL (incl. trie overhead)	11.0/10.3†
Average number of bytes per URL (excl. trie overhead)	7.0
Average number of bytes per link (only outlinks counted)	1.7
Number of documents processed	916,428
Total number of unique URL strings inserted	4,072,887
Compressed URL string space used (bytes)	28,529,733
SmallStringTrie *bigstring* branch overhead (bytes)	10,608,633
SmallStringTrie *jumptable* branch overhead (bytes)	5,996,360
Heap needed for *bigstring* attachment overhead (bytes)	1,790,230
Heap needed for *a_table* attachment overhead (bytes)	3,959,608
WebNode storage (bytes)	30,333,654
Heap needed for outlinks (incl. outlink count for each node)	21,093,064

TABLE 1. Summary of MarkovPR 1.1 memory requirements on Google dataset. Strings were compressed with a statistical arithmetic coder prior to insertion, otherwise no preprocessing was performed. † value expected once *jumptable* is optimized.

et al. report 10 bytes per URL and 3.4 bytes for each (bidirectional) link. Adler and Mitzenmacher concentrate exclusively on efficiently storing links, and report approximately 1.1 bytes per outlink in the best of cases, on the TREC WT2g dataset.

An important design decision in *MarkovPR 1.0* was to be able to read and process an unordered datastream of web documents in a single pass. By contrast, Suel and Yuan mention a long pruning and preprocessing stage of several hours, which includes completely sorting the input documents, and Adler and Mitzenmacher's results perform compression only on an already fully constructed web graph.

It is also important to realize thet the shape of the dataset has a great effect upon the storage requirements of the full graph. For example, Suel and Yuan report their results for a graph containing 11 million nodes and URLs, and 15 million links. By contrast, the Google dataset has 916,000 nodes, 4 million URLs and 12 million outlinks, and is not preprocessed in any significant way.

2. Baseline memory requirements

The initial implementation of *MarkovPR 1.0* is described in (Breyer, 2002). Here we summarize the memory requirements as inferred from a collection of nearly 1 million documents made available by Google, Inc.

Recall that an important design decision of the software consisted in reading unprocessed data repositories in a single pass to build the web graph. The two main reasons for this are that

- Meaningful preprocessing of the repositories, such as reordering the documents, is itself challenging for large datasets resulting from a web crawl. The results of the preprocessing must likely be stored separately and kept up to date with the actual data repositories.
- Speed of processing is an issue, since reading a large data repository is very slow and can be expected to occupy workstations a long time. Doubling or tripling this time due to preprocessing may be unacceptable for experimental purposes.

In its baseline implementation, *MarkovPR 1.0* consumes memory in two ways: one is the storage and query capability of document and link urls (this is approximately 80 Mb for Google's dataset, consisting of 4 million individual url strings), the other is the actual storage of the web graph (approximately 95 Mb, which includes 1 million nodes storing ID, date, occupation counts, scratch area, and pointers comprising all inlinks and outlinks, called tolinks and fromlinks respectively). Both these components should be optimizable, and here we shall do precisely this.

3. Optimizing url storage

In the baseline implementation, the document urls are kept in a linearized trie (Knuth, 1997). There are three basic data structures: a large string (*bigstring*) which contains the actual trie contents (overlapping, standardized url strings); a hashtable (*jumptable*) containing branching information needed for trie navigation; a hashtable (*nodetable*) containing pointers relating the inserted url to the corresponding *webnode*.

By splitting the trie into *bigstring* and *jumptable*, we obviate the need to reserve space for a branch destination pointer after each character of *bigstring*; instead we

need 8 bytes for each branch, consisting of a source pointer (the hash key) and a destination pointer (the hash value). Note that 4 byte pointers are suitable for addressing up to 4 Gb of data. The *jumptable* has approximately 4 million entries (one branch per unique inserted url string), while the nodetable only has slightly less than 1 million (one per document in the dataset).

3.1. **Optimizing the hashtables.** This is relatively easy, as follows: Since each entry in *jumptable* consists of two integer values, we should only store the actual number of bits needed to represent the values. Of course, this means we must also store the number *nb* of bits needed for one of the values at least.

Since the pointers range over the whole bigstring, in some cases 4 bytes (32 bits) will be needed (eg if *bigstring*'s size is on the order of a Gigabyte), and we therefore will need 5 bits for *nb*. Alternatively, a simpler system is to use the 8-th bit in each byte as a separation symbol between the two integers. The difference between these schemes is likely a few bits, but since we store each hashtable entry in an integral number of bytes anyway, it should not matter greatly.

An experiment with the first scheme on the dataset shows that approximately 1/3 of the space needed for *jumptable* can be gained, i.e. from 32 Mb to 22 Mb. However, we shall see in the next section a better approach.

3.2. **Optimizing the trie itself.** There are two things we can do to improve the memory requirements for the trie. One is to insert smaller strings, the other is to change the implementation of the trie. In the baseline implementation, the urls are standardized and uncompressed. By passing them through a compression filter which preserves the prefix structure of the strings, we can save some space for the trailing parts of the inserted strings, which otherwise only take up space.

3.2.1. *Inserting smaller strings.* Both arithmetic coding and LZW coding preserve in some sense the string prefixes, hence do not alter the final shape of the trie. By this we mean that if the strings s and s' share a nontrivial common prefix, then their compressed images $c(s)$ and $c(s')$ also share a nontrivial common prefix. We use arithmetic coding, and note that LZW is patent encumbered. After a few false starts, Mark Nelson's implementation (Nelson, 1991) was used for the compression.

One nonintuitive aspect is that we cannot use high performance adaptive dictionaries, since the coding of a given string would change during the lifetime of the graph construction. This poses a problem when searching for a given compressed url within the trie, as it may no longer be recognized.

Since arithmetic coding needs symbol statistics, these need to be gathered in a previous pass through the dataset. While this breaks the philosophy against multiple passes, it is relatively cheap (no significant disk space requirements) and could conceivably be done by the web crawler directly. For example an order-1 Markov model with 2-byte frequencies, consumes $2 \cdot 256^2$ bytes. The main problem is that without adaptability, the compression performance is unimpressive. Using the order-1 model on the Google dataset reduced the bigstring size from 42 Mb to 32 Mb. However, more sophisticated compression techniques will likely improve matters.

One simple possibility is to construct several order-1 models from different sections of the dataset, and for each url, try compressing with each model, keeping the best result as the one to be inserted in the trie. We also implemented a partial order-2 model which reduced the string size to approximately 28 Mb, at a dictionary cost of 4 Mb. Clearly this cost can be amortized over a larger dataset.

Incidentally, standard trie asymptotics (Knuth, 1997) apply here. For example, since the size of *bigstring* is simply the total number of internal nodes of the trie, we expect asymptotically $\mathbb{E}\,|bigstring| \approx n/H$, where n is the number of inserted strings and H is the entropy of the strings, assuming they have the Markov property. Better compression gives higher entropy, hence lower space requirements.

3.2.2. *A more efficient trie representation.* Besides inserting smaller strings, it is also possible to improve the representation of the branches. In the baseline implementation, each branch consumes 8 bytes, for a total of 32 Mb with 4 million distinct urls. Half of this consists in storing the branch starting point, which is conceptually redundant, since when we reach a branch, we only need to know the destination.

We can actually store the branch destinations directly mixed in with the *bigstring* data, if we are prepared to overwrite some of the data already stored there. We keep a copy of the overwritten data between the trie branches in *bigstring*, which

are normally simply appended in order of occurrence during construction. During trie traversal, we swap the data back in as needed. See the Appendix for details.

The gains achievable with this technique are quite impressive. First, we no longer store origin pointers (4 bytes), and second, destination pointers are naturally encoded in a variable number of bytes. A further optimization consists in storing the destination not as an address (i.e. character pointer), but as an offset from the branch origin, which is a much smaller number quite frequently.

There are two caveats: It isn't always possible to overwrite *bigstring* data reliably, so in some cases we store information externally, using the original *jumptable*. With our dataset, there are approximately $680,000$ entries in the *jumptable*, unlike the original 4 million.

Moreover, the new trie implementation only allows inserted strings to have a maximum length of 127.

The length constraint is not as problematic as it seems, since most inserted strings are only around half this size. For those strings exceeding 127 characters, we replace the end with a five character MD5 hash (see the discussion of hashing below). Table 1 summarizes the optimization gains due to some the above techniques (*jumptable* optimization is not implemented yet, but calculations show that the memory needed could be halved, resulting in an improved storage cost of 10.3 bytes per URL).

3.3. **Comparison with MD5 hashing.** It is interesting to discuss how our trie optimizations compare with a completely different scheme, namely that of storing the url strings in hashed form. The MD5 hash is an example of a readily available means of converting an arbitrarily long string s into a fixed size integer, $h(s)$. In the case of MD5, $h(s)$ is 16 bytes long.

First we note that any similarity between two strings s and s' is lost when hashing, so the resultant b-byte integers $h(s)$ and $h(s')$ cannot be stored more efficiently than side by side (as opposed to overlapping them, **Example 1.** in a trie). Theoretically, a good hash function will distribute the strings uniformly within the allowable range $0, \ldots, 2^b$, spreading two similar strings far apart. We can model this by assuming that $h(s)$ and $h(s')$ are independent whenever $s \neq s'$.

The following lemma is proved in ???

Lemma 1 (Birthday Lemma)**.** *Let* $\{h(s) : s = 1, \ldots, q\}$ *be a sequence of independent random variables, uniformly distributed over the range* $0, \ldots, N$*. Then for* $1 \leq q \leq \sqrt{2N}$*,*

$$0.3 \cdot \frac{q(q-1)}{N} \leq \mathbb{P}\big(h(s) \text{ are not all distinct}\big) \leq \frac{q(q-1)}{2N}.$$

Suppose we replace the trie with an array of hashed $q = 2^{8k}$ strings. Each string will occupy b bytes. By choosing b sufficiently large, we can guarantee a small enough probability of collision (when two distinct url strings hash to the same value). From Lemma 1, the probability of a collision will be less than $q(q-1)/2^{8b+1} \approx 2^{16k-8b-1}$, but more than $0.3 \cdot 2^{16k-8b}$. The largest web graphs collected so far consist of approximately 2 billion urls, so if we take $k = 4$, we see that $b = 8$ gives a probability of collision of at least 0.3, while if $b = 9$, the collision probability is at most 0.002.

It follows that the best hash functions cannot beat 9 bytes per url (assuming a total collection of 2 billion urls, which is not unreasonable in a distributed setting). If we assume the hash function is not optimal, this estimate should be revised upwards.

More generally, recall that in a distributed setting, we want to mark on each machine those locally stored urls which correspond to a node handled by another machine. This means that we cannot allow collisions between any two url strings within the whole distributed dataset. The size of the hash code is therefore determined by the size of the full dataset.

By contrast, a system using statistically based lossless compression of the urls requires an amount of memory per machine which depends only on the total number of urls stored locally on that machine. Hence increasing the number of machines or the size of the dataset does not affect compression performance on each individual machine. This difference in scaling behaviour becomes important for very large datasets.

In *MarkovPR 1.1*, we use hashing to truncate strings to a maximum length of 127 characters. Here we simply want to discuss the number of bytes required to prevent collisions. This problem is not directly covered by the previous discussion, as we need only prevent collisions between urls pertaining to a single subdirectory on a single web server. Again, we shall assume 2 billion url strings. Suppose the

string s has length greater than M, where $M = 127$. We want to hash the end of the string starting with character $M - r$, where $r = 2^{8b}$. By the above discussion, we can assume $r \leq 9$ already. Now it is clear that the unhashed portion of s is more than enough to contain the host server name and a large portion of the directory structure, so that collisions between hashed strings can only occur between urls on the same host in essentially the same directory. Let n_f be (an upper bound for) the number of HTML files collected from a single host, and n_h be (an upper bound for) the number of distinct host URLs stored on a single machine in the distributed cluster. Then from Lemma 1,

$$\begin{aligned}\mathbb{P}\big(\text{corrupted trie}\big) &\leq n_h \cdot \mathbb{P}(\text{hash collision on same host}) \\ &\leq 2^{\log_2(n_h)+2\log_2(n_f)-8b-1}\end{aligned}$$

Note that this bound is essentially linear in n_h. Assuming for example $\log_2(n_h) \leq 20$ and $\log_2(n_f) \leq 14$, it follows that that $b = 7$ will put the collision probability at less than 0.002. In *MarkovPR 1.1*, optimism compels us to use $b = 5$.

4. Optimizing the web graph

The baseline representation of the web graph requires nearly 100 Mb, which includes space for nodes containing a fixed size component (32 bytes) and two variable sized components (*tolinks* and *fromlinks* arrays). Each entry in the variable sized arrays is 4 bytes, and represents either a pointer to another node, or a dangling pointer (unresolved url).

4.1. **Optimizing the representation.** In the baseline implementation, several variables are packed together into each *WebNode*, as well as a list of *tolinks* and *fromlinks*. This is not very flexible in that it doesn't permit a unified approach towards associating various calculated properties to each *WebNode*. Future implementations of *MarkovPR* will be better served by implementing a properties manager for *WebNodes*. As a first step towards this more efficient system, we now store only an identifying ID and the outward pointing *tolinks* directly within the *WebNode*. The dual set of *fromlinks*, and other variables too, can be constructed on demand if required, and indexed by the *WebNode*'s ID.

More generally, it makes sense to store the *tolinks* at the *WebNode* since they represent essentially static information: As discussed in (Breyer, 2002), if the goal is to save physical memory, it makes no sense to prune or otherwise process the *tolinks* after the web graph has been fully constructed, and during construction this is impossible.

We aim therefore to compress as much as possible the *tolinks* during construction. This generally requires a tradeoff between memory size and access speed later during computations. Specifically, for each *WebNode*, we first sort the set of *tolinks* pointers, and compute the pointer differences, which we then store consecutively with a variable length encoding. The nth *tolink* can be retrieved by stepping through the first n differences and adding them.

This type of compression is quite natural given the growth pattern of the linearized trie. In particular, since all relevant *tolink* urls must be inserted (or found to be existing) during the construction of the *WebNode*, the pointer differences tend to be very small, which benefits compression. On the Google dataset, approximately half the total number of pointer differences can be represented in a single byte.

An interesting property of this *tolinks* representation, which may prove important in future investigations, is that these pointer differences are (partially) repeated every time two *WebNodes* share (partially) the same *tolinks*. This suggests that we can further compress the tolinks in case many web documents contain copied links. Various models for web graphs depend upon such copying properties, and in fact the graph compression results in (Adler and Mitzenmacher, 2001) depend on these.

One last aspect we want to mention here about the *tolinks* compression proposed above, which is implemented in *MarkovPR 1.1*, is that reconstructing the link pointers each time they are needed introduces a performance penalty, informally estimated to be a factor of 5. A simple caching scheme was introduced, which needs to be taken into account when constructing new page ranking samplers, but the benefits a very small, due mainly to the fact that the *PageRank*-type samplers do not generally visit the exact same *WebNode* often enough.

4.2. **Optimizing the dangling pointers.** The main difficulty with the dataset is the number of dangling pointers. Each such pointer arises as part of one or several nodes, but also as an entry in the trie. With the Google dataset, 3/4 of the urls

are not directly associated with a node, and on average half the outgoing links of each node are dangling.

Dangling pointers may or may not be resolved into leafnodes in the distributed setting, hence it is a bad idea to prune them early. Pruning these pointers after the web graph is fully constructed and connected to the other machines in the distributed case is somewhat pointless, since the reclaimed memory can't be used to create a bigger web graph (this would require multiple passes through the dataset). It is impractical to keep a detailed external list of censored urls for the purpose of preventing their addition in the web graph construction stage, but a flexible set of rules may allow heavy pruning. For example, we might decide to build only the **.edu* part of the web graph, which allows us to prune all links which leave that domain. This is a complicated subject which will need some research.

5. Appendix

The *SmallStringTrie* is a linearlized implementation of the classical trie data structure (Knuth, 1997, Section 6.3), in which the cost of keeping pointers to subtrees is reduced. We shall describe the structure informally on a simple example. Consider the following set of strings:

(1) `http://www.bu.edu/`
(2) `http://www.bu.edu/iscip/`
(3) `http://www.bu.edu/iscip/news.html`
(4) `http://www.bu.edu/iscip/perspective.html`
(5) `http://www.bu.edu/com/html/events.html`
(6) `http://www.bu.edu/com/html/faculty1.html`

In Figure 1, we represent the trie corresponding to these strings. Figure 2 shows a linearized version of Figure 1. The characters are stored consecutively in a character array called *bigstring*. Each arrow will be called a *branch*.

Navigation within the linearized trie is by means of two character pointers, p and s. The first always points into the trie, while the latter points to the string being inserted or searched. The pseudo-code to insert a new (null terminated) string s is shown in Listing 1 (searching for an existing string is similar).

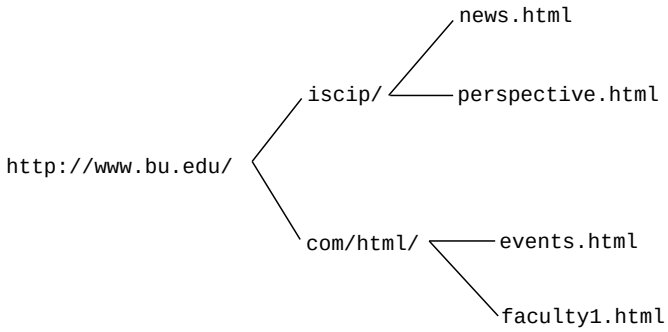

FIGURE 1. Trie structure prevents duplication of common url prefixes.

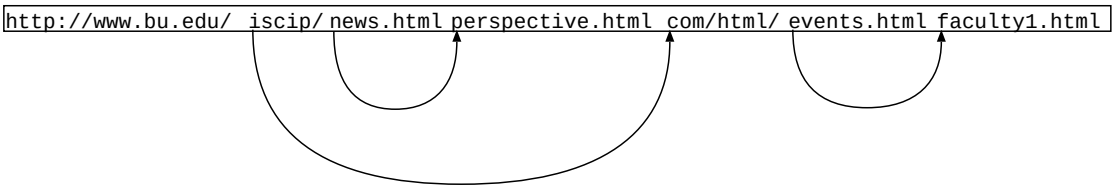

FIGURE 2. Linearised trie. Each arrow is a branch link, with origin k and destination v.

To store the branches outside the trie, we could keep pairs of origin and destination pointers. Alternatively, we must reserve space for a potential branch destination after each trie character. Both these methods are space consuming (especially the latter). Instead, the *SmallStringTrie* overwrites the trie data with a branch destination pointer at the location it is needed, placing a copy of the overwritten data at the branch destination. This technique will be called a *swap*.

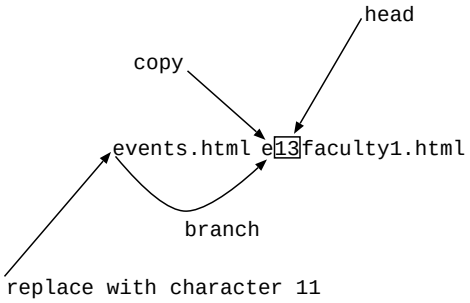

FIGURE 3. A swap operation creates a new branch (diagram omits heads).

LISTING 1

```
// p points to beginning of trie ,
// s points to beginning of inserted string
quit = false;
found = false;
while not quit {
  compare first character at p and s;
  if equal {
    p++; s++;
    if s is null {
      found = true;
      quit = true;
    }
  } else if p is a branch origin {
    p = branch destination;
  } else {
    found = false;
    quit = true;
  }
}
if not found {
  create another branch with origin p and
  destination the end of the trie ;
  append s at the branch destination;
}
```

Figure 3 illustrates the swap during insertion of the last string, *http://www.bu.edu/com/html/faculty1.html*. After the pointer p reaches the character e, there are no more branches, but s still points to the suffix *faculty1.html*, which must be appended. The length of the jump is known (11 characters, including the trailing null character), so the character e is copied at the end of the trie, and overwritten with a single byte integer whose value is 11, the length of the jump. After the

copied value of e comes the string pointed to by s. Clearly, the swap operation is reversible, i.e. the state of the trie before the swap is recoverable by overwriting with the copied data and fixing the head (defined below).

Since we no longer store the branch origin separately, we need a way to compute, for a given pointer p, if a branch originates from p (refer to Listing 1). The mechanism for this is as follows: At the head of the string, s, which we place after a branch destination, we insert a single character, called the *head*, which contains either the number of characters until the end of the appended string s (at first), or the number of characters until the next branch origin created within the appended string s (subsequently). The head can only contain a small number, which is why the *SmallStringTrie* is limited to strings of size 127, for we use one bit to mark whether the character count is to the end of s, or to the nearest branch origin.

The trie insertion and search pseudo-code in Listing 2 is substantially identical to Listing 1, except that, before the comparison between p and s, we check that there isn't a branch origin starting at p (by checking the distance from the nearest head). If there is, we read the branch destination and overwrite it with the saved data, which can be found at the branch destination. We then compare p with s to see if we take the new branch (whose destination is now known) or not.

To illustrate these ideas, we shall trace the state of the SmallTringTrie's *bigstring* during the successive insertions of the example urls (except for the first two, which we omit for readability. We begin with an empty *bigstring*. After inserting the url (3), we obtain

33`http://www.bu.edu/iscip/news.html`

The number in the box represents the head of the branch, and would be inserted as ASCII character 33. Next, we insert url (4), obtaining

24`http://www.bu.edu/iscip/`9`ews.htmln`**33**|**16**`perspective.html`

Here, a new branch was created 24 characters after the start of the string *http://ww...*, whose destination is 9 characters after its origin. The character n of the substring *news.html* is copied at the branch destination, and overwritten. The original head, which contained 33, is also copied and overwritten. Finally, the suffix *perspective.html* is written, together with an associated head containing its length, 16. We now insert the url (5), obtaining

LISTING 2

```
// h points to first head
// p = h + 1
// s points to beginning of inserted string
quit = false;
found = false;
while not quit {
  if p − h >= contents of h {
    j = integer encoded at p;
    overwrite integer at p with data at p + j;
    overwrite contents at h with head copy at p + j (after data);
  }
  // now p has data and h is updated
  compare first character of p and s;
  if equal {
    p++; s++;
    if s has zero length {
      found = true;
      quit = true;
    }
  } else if p is a branch origin {
    h = branch destination (after swap data);
    p = h + 1;
  } else {
    found = false;
    quit = true;
  }
}
undo all swaps in reverse order;
if not found {
  create another branch with origin p
  and destination the end of the trie;
  append another head containing size of s;
  append s;
}
```

18 http://www.bu.edu/ 34 scip/ 9 ews.htmln **33** ···

··· **16** perspective.htmli 24 **19** com/html/events.html

Similarly, insertion of url (6) gives

18 http://www.bu.edu/ 34 scip/ 9 ews.htmln **33** ···

··· **16** perspective.htmli 24 ···

··· 9 com/html/ 11 vents.htmle **19** **13** faculty1.html

For more specific implementation details, including the boundary case when there is no room to overwrite existing data at a new branch origin, we refer the reader to the *MarkovPR 1.1* source code.

The memory cost of the *SmallStringTrie* implementation is as follows: Each inserted string contributes a single branch, hence an overhead of the space needed to store a branch destination only, as a distance from the branch origin. Moreover, each branch requires a copy of the head. Otherwise, the memory requirements are exactly the same as in the trie, since each string's trailing null is no longer needed, being replaced by the nearest head value.

References

[1] Adler, M. and Mitzenmacher, M. (2001) Towards compressing web graphs. In *Proceedings of the IEEE data compression conference (DCC)* March 2001.

[2] Breyer, L.A. (2002) Markovian Page Ranking Distributions: Some Theory And Simulations.

[3] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A. and Wiener, J. (2000) Graph structure in the web: experiments and models. In *9th Int. World Wide Web Conference, 2000.*

[4] Knuth, D. (1997) *The Art Of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley.

[5] Nelson, M. (1991) *Arithmetic Coding + Statistical Modeling = Data Compression* Dr. Dobb's Journal, Feb. 1991.

[6] Suel, T. and Yuan, J. (2001) Compressing the Graph Structure of the Web. In *Proceedings of the IEEE data compression conference (DCC)* March 2001..

E-mail address: lairdlbreyer.com