A Fast and Compact Web Graph Representation *

Francisco Claude and Gonzalo Navarro

Department of Computer Science, Universidad de Chile
{fclaude,gnavarro}@dcc.uchile.cl

Abstract. Compressed graphs representation has become an attractive research topic because of its applications in the manipulation of huge Web graphs in main memory. By far the best current result is the technique by Boldi and Vigna, which takes advantage of several particular properties of Web graphs. In this paper we show that the same properties can be exploited with a different and elegant technique, built on Re-Pair compression, which achieves about the same space but much faster navigation of the graph. Moreover, the technique has the potential of adapting well to secondary memory. In addition, we introduce an approximate Re-Pair version that works efficiently with limited main memory.

1 Introduction

A compressed data structure, besides answering the queries supported by its classical (uncompressed) counterpart, uses little space for its representation. Nowadays this kind of structures is receiving much attention because of two reasons: (1) the enormous amounts of information digitally available, (2) the ever-growing speed gaps in the memory hierarchy. As an example of the former, the graph of the static indexable Web was estimated in 2005 to contain more than 11.5 billion nodes [12] and more than 150 billion links. A plain adjacency list representation would need around 600 GB. As an example of (2), access time to main memory is about one million times faster than to disk. Similar phenomena arise at other levels of memory hierarchy. Although memory sizes have been growing fast, new applications have appeared with data management requirements that exceeded the capacity of the faster memories. Because of this scenario, it is attractive to design and use compressed data structures, even if they are several times slower than their classical counterpart. They will run much faster anyway if they fit in a faster memory.

In this scenario, compressed data structures for graphs have suddenly gained interest in recent years, because a graph is a natural model of the Web structure. Several algorithms used by the main search engines to rank pages, discover communities, and so on, are run over those Web graphs. Needless to say, relevant Web graphs are huge and maintaining them in main memory is a challenge, especially if we wish to access them in compressed form, say for navigation purposes.

* Partially funded by a grant from Yahoo! Research Latin America.

As far as we know, the best results in practice to compress Web graphs such that they can be navigated in compressed form are those of Boldi and Vigna [6]. They exploit several well-known regularities of Web graphs, such as their skewed in- and out-degree distributions, repetitiveness in the sets of outgoing links, and locality in the references. For this sake they resort to several ad-hoc mechanisms such as node reordering, differential encoding, compact interval representations and references to similar adjacency lists.

In this paper we present a new way to take advantage of the regularities that appear in Web graphs. Instead of different ad-hoc techniques, we use a uniform and elegant technique called Re-Pair [19] to compress the adjacency lists. As the original linear-time Re-Pair compression requires much main memory, we develop an approximate version that adapts to the available space and can smoothly work on secondary memory thanks to its sequential access pattern. This method can be of independent interest. Our experimental results over different Web crawls show that our method achieves space comparable to that of Boldi and Vigna, yet our navigation is several times faster.

2 Related Work

Let us consider graphs $G = (V, E)$, where V is the set of vertices and E is the set of edges. We call $n = |V|$ and $e = |E|$ in this paper. Standard graph representations such as the incidence matrix and the adjacency list require $n(n-1)/2$ and $2e \log n$ bits, respectively, for undirected graphs. For directed graphs the numbers are n^2 and $e \log n$, respectively[1]. We call the *neighbors* of a node $v \in V$ those $u \in V$ such that $(v, u) \in E$.

The oldest work on graph compression focuses on undirected unlabeled graphs. The first result we know of [30] shows that planar graphs can be compressed into $O(n)$ bits. The constant factor was later improved [17], and finally a technique yielding the optimal constant factor was devised [14]. Results on planar graphs can be generalized to graphs with constant *genus* [20]. More generally, a graph with genus g can be compressed into $O(g + n)$ bits [10]. The same holds for a graph with g *pages*. A page is a subgraph whose nodes can be written in a linear layout so that its edges do not cross. Edges of a page hence form a nested structure that can be represented as a balanced sequence of parentheses.

Some classes of planar graphs have also received special attention, for example trees, triangulated meshes, triconnected planar graphs, and others [15, 17, 13, 28]. For dense graphs, it is shown that little can be done to improve the space required by the adjacency matrix [23].

The above techniques consider just the compression of the graph, not its access in compressed form. The first compressed data structure for graphs we know of [16] requires $O(gn)$ bits of space for a g-page graph. The neighbors of a node can be retrieved in $O(\log n)$ time each (plus an extra $O(g)$ complexity for the whole query). The main idea is again to represent the nested edges using

[1] In this paper logarithms are in base 2.

parentheses, and the operations are supported using succinct data structures that permit navigating a sequence of balanced parentheses. The retrieval was later improved to constant time by using improved parentheses representations [22], and also the constant term of the space complexity was improved [9]. The representation also permits finding the degree (number of neighbors) of a node, as well as testing whether two nodes are connected or not, in $O(g)$ time.

All those techniques based on number of pages are unlikely to scale well to more general graphs, in particular to Web graphs. A more powerful concept that applies to this type of graph is that of graph *separators*. Although the separator concept has been used a few times [10, 14, 8] (yet not supporting access to the compressed graph), the most striking results are achieved in recent work [5, 4]. Their idea is to find graph components that can be disconnected from the rest by removing a small number of edges. Then, the nodes within each component can be renumbered to achieve smaller node identifiers, and only a few external edges must be represented.

They [4] apply the separator technique to design a compressed data structure that gives constant access time per delivered neighbor. They carefully implement their techniques and experiment on several graphs. In particular, on a graph of 1 million (1M) nodes and 5M edges from the Google programming contest[2], their data structures require 13–16 bits per edge (bpe), and work faster than a plain uncompressed representation using arrays for the adjacency lists. It is not clear how these results would scale to larger graphs, as much of their improvement relies on smart caching, and this effect should vanish with real Web graphs.

There is also some work specifically aimed at compression of Web graphs [7, 1, 29, 6]. In this graph the (labeled) nodes are Web pages and the (directed) edges are the hyperlinks. Several properties of Web graphs have been identified and exploited to achieve compression:

Skewed distribution: The indegrees and outdegrees of the nodes distribute according to a power law, that is, the probability that a page has i links is $1/i^\theta$ for some parameter $\theta > 0$. Several experiments give rather consistent values of $\theta = 2.1$ for incoming and $\theta = 2.72$ for outgoing links [2, 7].

Locality of reference: Most of the links from a site point within the site. This motivates in [3] the use of lexicographical URL order to list the pages, so that outgoing links go to nodes whose position is close to that of the current node. Gap encoding techniques are then used to encode the differences among consecutive target node positions.

Similarity of adjacency lists: Nodes close in URL lexicographical order share many outgoing links [18, 6]. This permits compressing them by a reference to the similar list plus a list of edits. Moreover, this translates into source nodes pointing to a given target node forming long intervals of consecutive numbers, which again permits easy compression.

In [29] they partition the adjacency lists considering popularity of the nodes, and use different coding methods for each partition. A more hierarchical view

[2] `www.google.com/programming-contest`, not anymore available.

of the nodes is exploited in [26]. In [1, 27] they take explicit advantage of the similarity property. A page with similar outgoing links is identified with some heuristic, and then the current page is expressed as a reference to the similar page plus some edit information to express the deletions and insertions to obtain the current page from the referenced one. Finally, probably the best current result is from [6], who build on previous work [1, 27] and further engineer the compression to exploit the properties above.

Experimental figures are not always easy to compare, but they give a reasonable idea of the practical performance. Over a graph with 115M nodes and 1.47 billion (1.47G) edges from the Internet Archive, [29] require 13.92 bpe (plus around 50 bits per node, bpn). In [27], over a graph of 61M nodes and 1G edges, they achieve 5.07 bpe for the graph. In [1] they achieve 8.3 bpe (no information on bpn) over TREC-8 Web track graphs (WT2g set), yet they cannot access the graph in compressed form. In [7] they require 80 bits per node plus 27.2 bpe (and can answer reverse neighbor queries as well).

By far the best figures are from [6]. For example, they achieve space close to 3 bpe to compress a graph of 118M nodes and 1G link from WebBase[3]. This space, however, is not sufficient to access the graph in compressed form. An experiment including the extra information required for navigation is carried out on a graph of 18.5M nodes and 292M links, where they need 6.7 bpe to achieve access times below the microsecond. Those access times are of the same order of magnitude of other representations [29, 26, 27]. For example, the latter reports times around 300 nanoseconds per delivered edge.

A recent proposal [24] advocates regarding the adjacency list representation as a text sequence and use compressed text indexing techniques [25], so that neighbors can be obtained via text decompression and reverse neighbors via text searching. The concept and the results are interesting but not yet sufficiently competitive with those of [6].

3 Re-Pair and Our Approximate Version

Re-Pair [19] is a phrase-based compressor that permits fast and local decompression. It consists of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol, until no more replacements are convenient. This technique was recently used in [11] for compressing suffix arrays. More precisely, Re-Pair over a sequence T works as follows:

1. It identifies the most frequent pair ab in T
2. It adds the rule $s \rightarrow ab$ to a dictionary R, where s is a new symbol that does not appear in T.
3. It replaces every occurrence of ab in T by s.[4]
4. It iterates until every pair in T appears once.

[3] `www-diglib.stanford.edu/~testbed/doc2/WebBase/`

[4] As far as possible, e.g. one cannot replace both occurrences of aa in aaa.

Let us call C the resulting text (i.e., T after all the replacements). It is easy to expand any symbol c from C in time linear on the expanded data (that is, optimal): We expand c using rule $c \rightarrow c'c''$ in R, and continue recursively with c' and c'', until we obtain the original symbols of T. In [11] they propose a new way of compressing the dictionary R which further reduces the space. This compression makes it worthy to replace pairs that appear just twice in the text.

Despite its quadratic appearance, Re-Pair can be implemented in linear time [19]. However, this requires several data structures to track the pairs that must be replaced. This is usually problematic when applying it to large sequences, see for example [31]. Indeed, it was also a problem when using it over suffix arrays [11], where a couple of techniques were proposed to run with restricted memory. A first one was an $O(n \log n)$ time exact method (meaning that it obtained the same result as the algorithm as described), which was in practice extremely slow. A second one was an approximate algorithm (which does not always choose the most frequent pair to replace), which was much faster and lost some compression, yet it applies only to suffix arrays.

We present now an alternative approximate method that (1) works on any sequence, (2) uses as little memory as desired on top of T, (3) given an extra memory to work, can trade accurateness for speed, (4) is able to work smoothly on secondary memory due to its sequential access pattern.

We describe the method assuming we have $M > |T|$ main memory available. If this is not the case, we can anyway run the algorithm by maintaining T on disk since, as explained, we will access it sequentially (performing several passes).

We place T inside the bigger array of size M, and use the remaining space as a (closed) hash table H of size $|H| = M - |T|$. Table H stores unique pairs of symbols ab occurring in T, and a counter of the number of occurrences it has in T. The key ab is represented as a single integer by its position in T (any occurrence works). We traverse $T = t_1 t_2 \ldots$ sequentially and insert all the pairs $t_i t_{i+1}$ into H. If, at some point, the table surpasses a load factor $0 < \alpha < 1$, we do not insert new pairs anymore, yet we keep traversing of T to increase the counters of already inserted pairs.

After the traversal is completed, we scan H and retain the k most frequent pairs from it, for some parameter $k \geq 1$. A heap of k integers is sufficient for this purpose. Those pairs will be simultaneously replaced in a second pass over T. For this sake we must consider that some replacements may invalidate others, for example we cannot replace both ab and bc in abc. Some pairs can have so many occurrences invalidated that they are not worthy of replacement anymore (especially at the end, when even the most frequent pairs occur a few times).

The replacement proceeds as follows. We empty H and insert only the k pairs to be replaced. This time we associate to each pair a field *pos*, the position of its first occurrence in T. This value is *null* if we have not yet seen any occurrence in this second pass, and *proceed* if we have already started replacing it. We now scan T and use H to identify pairs that must be replaced. If pair ab is in H and its *pos* value is *proceed*, we just replace ab by sz, where s is the new symbol for pair ab and z is an invalid entry. If, instead, pair ab already has a first position

recorded in *pos*, and we read this position in T and it still contains ab (after possible replacements that occurred after we saw that position), then we make both replacements and set the *pos* value to *proceed*. Otherwise, we set the *pos* value of pair ab to the current occurrence we are processing. This method ensures that we create no new symbols s that will appear just once in T.

After this replacement, we compact T by deleting all the z entries, and restart the process. As now T is smaller, we can have a larger hash table of size $|H| = M - |T|$. The traversal of T, regarded as a circular array, will now start at the point where we stopped inserting pairs in H in the previous stage, to favor a uniform distribution of the replacements.

Assume the exact method carries out r replacements. This approximate method can carry out r replacements (achieving hopefully similar compression) in time $O(\lceil r/k \rceil (n + h \log k))$ average time, where $h = |H| = O(n)$. Thus we can trade time for accurateness by tuning k. This analysis, however, is approximate, as some replacements could be invalidated by others and thus we cannot guarantee that we carry out k of them per round. Yet, the analysis is useful to explain the space/time tradeoff involved in the choice of k.

Note that even $k = 1$ does not guarantee that the algorithm works exactly as Re-Pair, as we might not have space to store all the different pairs in H. In this respect, it is interesting that we become more accurate (thanks to a larger H) for the later stages of the algorithm, as by that time the frequency distribution is flatter and more precision is required to identify the best pairs to replace.

As explained, the process works well on disk too. This time T is on disk and table H occupies almost all the main memory, $|H| \approx M$. In H we do not store the position of pair ab but instead ab explicitly, to avoid random accesses to T. The other possible random access is the one where we check *pos*, the first occurrence of a pair ab, when replacing T. Yet, we note that there are at most k positions in T needing random access at any time, so a buffer of k disk pages totally avoids the cost of those accesses. That is, we maintain in main memory the disk blocks containing the position *pos* of each pair to be replaced. When *pos* changes we can discard the disk block from main memory and retain the new one instead (which is the block we are processing). It is possible, however, that we have to write back those pages to disk, but this occurs only when we replace the first occurrence of a pair ab, that is, when *pos* changes from a position to the value *proceed*. This occurs at most k times per stage.

Thus the worst-case I/O cost of this algorithm is $O(\lceil r/k \rceil (n/B + k)) = O(\lceil r/k \rceil \; n/B + r + k)$, where B is the disk block size (again, this is is an approximation).

4 A Compressed Graph Representation

Let $G = (V, E)$ be the graph we wish to compress and navigate. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of nodes in arbitrary order, and $adj(v_i) = \{v_{i,1}, v_{i,2}, \ldots v_{i,a_i}\}$ the set of neighbors of node v_i. Finally, let $\overline{v_i}$ be an alternative identifier

for node v_i. We represent G by the following sequence:

$$T = T(G) = \overline{v_1}\ v_{1,1}\ v_{1,2} \ldots v_{1,a_1}\ \overline{v_2}\ v_{2,1}\ v_{2,2} \ldots v_{2,a_2}\ \ldots\ \overline{v_n}\ v_{n,1}\ v_{n,2} \ldots v_{1,a_n}$$

so that $v_{i,j} < v_{i,j+1}$ for any $1 \leq i \leq n$, $1 \leq j < a_i$. This is essentially the concatenation of all the adjacency lists with separators that indicate the node each list belongs to.

Applying Re-Pair to this representation $T(G)$ has several advantages:

- Re-Pair permits fast local decompression, as it is a matter of extracting successive symbols from C (the compressed T) and expanding them using the dictionary of rules R.
- This works also very well if $T(G)$ must be anyway stored in secondary memory because the accesses to C are local and sequential, and moreover we access fewer disk blocks because it is a compressed version of T. This requires, however, that R fits in main memory, which can be forced at compression time, at the expense of losing some compression ratio.
- As the symbols $\overline{v_i}$ are unique in T, they will not be replaced by Re-Pair. This guarantees that the beginning of the adjacency list of each v_i will start at a new symbol in C, so that we can decompress it in optimal time $O(|adj(v_j)|)$ without decompressing unnecessary symbols.
- If there are similar adjacency lists, Re-Pair will spot repeated pairs, therefore capturing them into shorter sequences in C. Actually, assume $adj(v_i) = adj(v_j)$. Then Re-Pair will end up creating a new symbol s which, through several rules, will expand to $adj(v_i) = adj(v_j)$. In C, the text around those nodes will read $\overline{v_i} s \overline{v_{i+1}} \ldots \overline{v_j} s \overline{v_{j+1}}$. Even if those symbols do not appear elsewhere in $T(G)$, the compression method for R developed in [11] will represent R using $|adj(v_i)|$ numbers plus $1 + |adj(v_i)|$ bits. Therefore, in practice we are paying almost the same as if we referenced one adjacency list from the other. Thus we achieve, with a uniform technique, the result achieved in [6] by explicit techniques such as looking for similar lists in an interval of nearby nodes.
- Even when the adjacency lists are not identical, Re-Pair can take partial advantage of their similarity. For example, if we have *abcde* and *abde*, Re-Pair can transform them to scs' and ss', respectively. Again, we obtain automatically what in [6] is done by explicitly encoding the differences using bitmaps and other tools.
- The locality property is not exploited by Re-Pair, unless its translates into similar adjacency lists. This, however, makes our technique independent of the numbering. In [6] it is essential to be able of renumbering the nodes according to site locality. Despite this is indeed a clever numbering for other reasons, it is possible that renumbering is forbidden if the technique is used inside another application. However, we show next a way to exploit locality.

The representation $T(G)$ we have described is useful for reasoning about the compression performance, but it does not give an efficient method to know where a list $adj(v_i)$ begins. For this sake, after compressing $T(G)$ with Re-Pair,

we remove all the symbols $\overline{v_i}$ from the compressed sequence C (as explained, those symbols must remain unaltered in C). Using exactly the same space we have gained with this removal, we create a table that, for each node v_i, stores a pointer to the beginning of the representation of $adj(v_i)$ in C. With it, we can obtain $adj(v_i)$ in optimal time for any v_i.

If we are allowed to renumber the nodes, we can exploit the locality property in a subtle way. We let the nodes be ordered and numbered by their URL, and every adjacency list encoded using differential encoding. The first value is absolute and the rest represents the difference to the previous value. For example the list 4 5 8 9 11 12 13 is encoded as 4 1 3 1 2 1 1.

Differential encoding is usually a previous step to represent small numbers with fewer bits. We do not want to do this as it hampers decoding speed. Our main idea to exploit differential encoding is that, if every node tends to have local links, there will be many small differences we could exploit with Re-Pair, say pairs like $(1, 1)$, $(1, 2)$, $(2, 1)$, etc. We also present results for this variant and show that the compression is improved at the expense of some extra decompression.

5 Experimental Results

The experiments were run on a Pentium IV 3.0 GHz with 4GB of RAM using Gentoo GNU/Linux with kernel 2.6.13 and `g++` with `-O9` and `-DNDEBUG` options. The compression ratios r we show are the compressed file size as a percentage of the uncompressed file size.

We first study the performance of our approximate technique described in Section 3, as compared to the original technique. Table 1 shows the results for different M (amount of main memory for construction) and k parameters, over a 400 MB suffix array built from a 100 MB XML file (see [11]). The same file with the method proposed in [11] achieves $r = 20.08\%$ after 7.22 hours. The approximate version that works only over suffix arrays achieves 21.3% in 3 minutes.

As it can be seen, our approximate method obtains compression ratios reasonably close to the original one, while being practical in time and RAM for construction. In the following we only use the approximate method, as our graphs are much larger and the exact method does not run.

k	M (MB)	r %	time(min)	k	M (MB)	r %	time(min)
10,000,000	1126	23.41%	22	10,000,000	891	27.40%	21
5,000,000	930	23.25%	13	5,000,000	763	27.86%	12
1,000,000	840	22.68%	13	1,000,000	611	29.42%	14
500,000	821	22.44%	18	500,000	592	28.30%	21
100,000	805	22.03%	59	100,000	576	30.49%	67

Table 1. Compression ratios and times for different memory usage for construction M, and parameter k.

We now study our graph compression proposal (Section 4). Fig. 1 shows the results for four Web crawls, all downloaded from `http://law.dsi.unimi.it/`. `UK` is a graph with 18,520,486 nodes and 298,113,762 edges, `EU` has 862,664 nodes and 19,235,140 edges, `Arabic` has 22,744,080 nodes and 639,999,458 edges, and `Indochina` has 7,414,866 nodes and 194,109,311 edges.

We show on the left side the behavior of our Re-Pair-based method with and without differential encoding, compared to Boldi & Vigna's implementation [6] run on our machine with different space/time tradeoffs. The space is measured in bits per edge (bpe), where the total space cost is divided by the number of edges. The implementation of Boldi & Vigna gives a bpe measure that is consistent with the sizes of the generated files. However, their process (in Java) needs more memory to run. This could suggest that they actually need to build more structures that are not stored on the file, but this is difficult to quantify because of other space overheads that come from Java itself and from the WebGraph framework their code is inside. To account for this, we draw a second line that shows the minimum amount of RAM needed for their process to run. In all cases, however, the times we show are obtained with the garbage collector disabled and sufficient RAM to let the process achieve maximum speed. Although our own code is in C++, we found that the Java compiler achieves very competitive results (in unrelated tests over a similar code).

On the right side we compare our method with two fast uncompressed representations: a plain one using 32-bit integers to represent the adjacency lists, and a compact representation using $\lceil \log_2 n \rceil$ bits for every link and $\lceil \log_2 m \rceil$ for every node (to point inside the adjacency list).

The results show that our method is a very competitive alternative to Boldi & Vigna's technique, which is currently the best by a wide margin for Web graphs. In all cases, our method gives a comparable amount of space. Moreover, using the same amount of space, our method is always faster (usually twice as fast, even considering their best line). In addition, one of our versions does not impose any particular node numbering.

Compared to an uncompressed graph representation, our method is also a very interesting alternative. It is 3–5 times smaller than the compact version and 2–3 times slower than it; and it is 4–6 times smaller than the the plain version and 3–6 times slower. In particular, a graph like `Arabic` needs 2.4 GB of RAM with a plain representation, whereas our compressed version requires only 420 MB of RAM. This can be easily manipulated in a normal 1 GB machine, whereas the plain version would have to resort to disk.

6 Conclusions

We have presented a graph compression method that takes advantage of similarity between adjacency lists by using Re-Pair [19], a phrase-based compressor. The results over different Web crawls show that our method achieves compression ratios similar to the best current schemes [6], while being significantly faster to navigate the compressed graph. Our scheme adapts well to secondary memory,

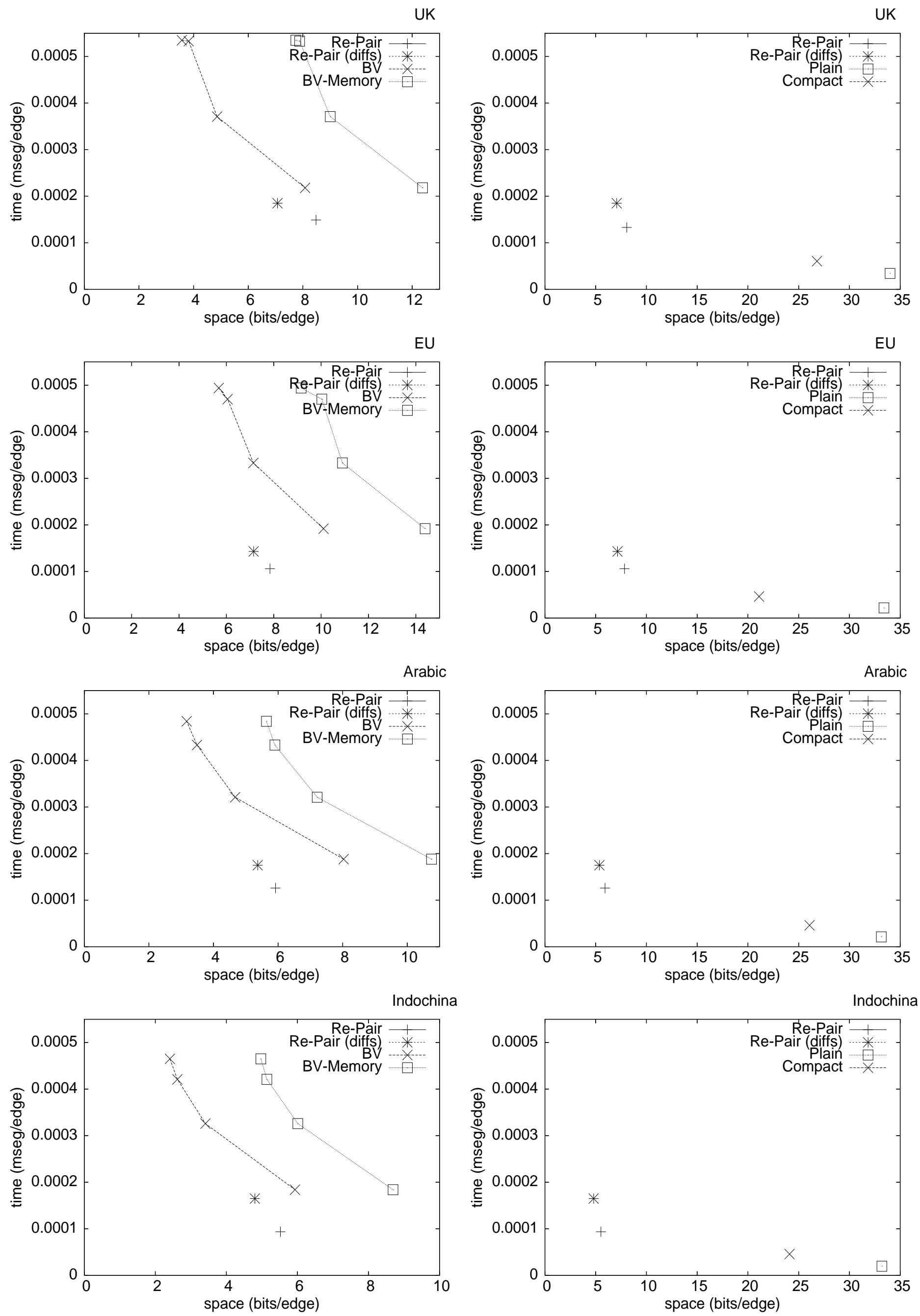

Fig. 1. Space and time to find neighbors for different graph representations, over different Web crawls. BV-Memory represents the minimum heap space needed by the process to run.

where it can take fewer accesses to disk than its uncompressed counterpart for navigation. In passing, we developed an efficient approximate version of Re-Pair, which also works well on secondary memory. Our work opens several interesting lines for future work:

1. More thorough experiments, considering also the case where the construction and/or the navigation must access secondary memory.
2. Thorough exploration of the performance of our approximate Re-Pair method by itself. Apart from studying it in more general scenarios, we are considering tuning it in different ways. For example we could use a varying k across the compression stages.
3. Further study of the compression format itself and tradeoffs. For example it is possible to compress sequence C with a zero-order compressor (the zero-order entropy of our C sequences tells that its size could be reduced to 61%-77%), although expansion of symbols in C will be slower. Another tradeoff is obtained by replacing the vector of pointers from each v_j to its list in C by a bitmap of $|C|$ bits that mark the beginning of the lists. A $select(j)$ operation (which gives the position of the j-th bit set [21]) over this bitmap would give the position of $adj(v_j)$ in C. This is slower than a direct pointer but will usually save space. For example we would save 1.0 bits/edge in the `UK` crawl (estimated without an implementation).
4. Combine the current representation with the ideas advocated in [24], so as to have a kind of self-index which, with some overhead over the current representation, would be able of finding reverse neighbors and answer other queries such as indegree and outdegree of a node, presence of a specific link, and so on[5].

References

1. M. Adler and M. Mitzenmacher. Towards compressing Web graphs. In *Proc. IEEE DCC*, pages 203–212, 2001.
2. W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proc. ACM STOC*, pages 171–180, 2000.
3. K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The Connectivity Server: Fast access to linkage information on the web. In *Proc. WWW*, pages 469–477, 1998.
4. D. Blandford. *Compact data structures with fast queries.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2006. Also as TR CMU-CS-05-196.
5. D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. SODA*, pages 579–588, 2003.
6. P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proc. WWW*, pages 595–602, 2004.
7. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *J. Computer Networks*, 33(1–6):309–320, 2000. Also in *Proc. WWW9*.

[5] This line of work is in cooperation with P. Ferragina and R. Venturini, University of Pisa.

8. D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsos. Fully automatic cross-associations. In *Proc. ACM SIGKDD*, 2004.
9. R. Chuang, A. Garg, X. He, M.-Y. Kao, and H.-I. Lu. Compact encodings of planar graphs with canonical orderings and multiple parentheses. In *LNCS v. 1443*, pages 118–129, 1998.
10. N. Deo and B. Litow. A structural approach to graph compression. In *Proc. MFCS Workshop on Communications*, pages 91–101, 1998.
11. R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. CPM*, LNCS 4580, pages 216–227, 2007.
12. A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Proc. WWW*, 2005.
13. X. He, M.-Y. Kao, and H.-I. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *J. Discrete Mathematics*, 12(3):317–325, 1999.
14. X. He, M.-Y. Kao, and H.-I. Lu. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM J. Comput.*, 30:838–846, 2000.
15. A. Itai and M. Rodeh. Representation of graphs. *Acta Informatica*, 17:215–219, 1982.
16. G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.
17. K. Keeler and J. Westbook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58:239–252, 1995.
18. R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large scale knowledge bases from the Web. In *Proc. VLDB*, 1999.
19. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.
20. H.-I. Lu. Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits. In *Proc. SODA*, pages 223–224, 2002.
21. I. Munro. Tables. In *Proc. FSTTCS*, LNCS 1180, pages 37–42, 1996.
22. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. FOCS*, pages 118–126, 1997.
23. M. Naor. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics*, 28(303–307), 1990.
24. G. Navarro. Compressing web graphs like texts. Technical Report TR/DCC-2007-2, Dept. of Computer Science, University of Chile, 2007.
25. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
26. S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proc. ICDE*, 2003.
27. K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The LINK database: Fast access to graphs of the Web. Technical Report 175, Compaq Systems Research Center, Palo Alto, CA, 2001.
28. J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization*, 5(1):47–61, 1999.
29. T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. IEEE DCC*, pages 213–222, 2001.
30. G. Turán. Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.
31. R. Wan. *Browsing and Searching Compressed Documents*. PhD thesis, Dept. of Computer Science and Software Engineering, University of Melbourne, 2003.