Fast and Compact Web Graph Representations *

Francisco Claude Gonzalo Navarro

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

Abstract

Compressed graph representations, in particular for Web graphs, have become an attractive research topic because of their applications in the manipulation of huge 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 that builds on on grammar-based compression. In particular, we focus on Re-Pair and on Ziv-Lempel compression, which achieve much faster navigation of the graph while using the same (and sometimes even less) space. Moreover, the technique adapts well to secondary memory. As a byproduct, 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 [18] and more than 150 billion links. A plain adjacency list representation of this graph 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. Distributed computation has been claimed to be a solution to those problems. However, access to a remote memory involves a waiting time that is closer to that of a disk access than to a local memory access. 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 manage to fit in a faster memory.

In this scenario, compressed data structures for graphs have suddenly gained interest in recent years, because a (directed) 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

*Partially funded by Yahoo! Research project "Compact Data Structures". An earlier partial version of this work appeared in *Proc. SPIRE 2007*.

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.

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 [8]. 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 [26] 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.

We also show that other grammar-based compressors can be used instead of Re-Pair, as long as they are able of efficiently extracting snippets from a sequence and of handling large alphabets. In particular, we modify the Ziv-Lempel variant called LZ78 [41] in order to achieve random access. LZ78 does not compress as much as our Re-Pair variants, yet it is slightly faster to extract snippets.

Our experimental results over different Web crawls show that our methods require space very similar (and sometimes slightly lower) than that of Boldi and Vigna. Moreover, when the latter is tuned to use the same space as our methods, ours are 2–3 times faster. Compared to a plain graph representation, ours is shown to be up to 13 times smaller, which largely increases the chance to fit very large graphs in main memory. Even when this is not possible, our secondary memory variant is also extremely interesting: For example, based on our results, we extrapolate that the 600 GB estimation for the whole static indexable Web would be accessed up to 5 times faster on disk thanks to compression, requiring just 4 to 8 GB of RAM. This is a perfectly reasonable requirement nowadays.

2 Related Work

2.1 Graph Representations

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 $m = |E|$ in this paper. Standard graph representations such as the incidence matrix and the adjacency list require $n(n-1)/2$ and $2m \log n$ bits[1], respectively, for undirected graphs. For directed graphs the numbers are n^2 and $m \log n$, respectively. 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 [38] shows that planar graphs can be compressed into $O(n)$ bits. The constant factor was later improved [24], and finally a technique yielding the optimal constant factor was devised [20]. Results on planar graphs can be generalized to graphs with constant *genus* [27]. More generally, a graph with genus g can be compressed into $O(g + n)$ bits [13]. 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

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

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 [22, 24, 19, 36]. For dense graphs, it is shown that little can be done to improve the space required by the adjacency matrix [30].

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 [23] 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 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 [29], and also the constant term of the space complexity was improved [11]. 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 [13, 20, 10] (yet not supporting access to the compressed graph), the most striking results are achieved in recent work [7, 6]. 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 [6] 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 [9, 1, 37, 8]. 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 in- and out-degrees 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, 9].

Locality of reference: Most of the links from a site point within the site. This motivates 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 [5]. Gap encoding techniques are then used to encode the differences among consecutive target node positions.

Similarity of adjacency lists: Nodes tend to share many outgoing links with some other nodes [25, 8]. This permits compressing them by a reference to the similar list plus a list of edits.

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

Suel and Yuan [37] partition the adjacency lists considering popularity of the nodes, and use different coding methods for each partition. A more hierarchical view of the nodes is exploited by Raghavan and Garcia-Molina [33]. Different authors [1, 35] 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 encode the deletions and insertions needed to obtain the current page from the referenced one. Finally, probably the best current result is from Boldi and Vigna [8], who build on previous work [1, 35] 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 performances. Over a graph with 115M nodes and 1.47 billion (1.47G) edges from the Internet Archive, Suel and Yuan [37] require 13.92 bpe (plus around 50 bits per node, bpn). Randall et al. [35], over a graph of 61M nodes and 1G edges, achieve 5.07 bpe for the graph. Adler and Mitzenmatcher [1] 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. Broder at el. [9] require 80 bits per node plus 27.2 bpe (and can answer reverse neighbor queries as well).

By far the best figures are from Boldi and Vigna [8]. 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 efficiently 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 in their machine. Those access times are of the same order of magnitude of other representations [37, 33, 35]. For example, the latter reports times around 300 nanoseconds per delivered edge.

A recent proposal [31] advocates regarding the adjacency list representation as a text sequence and use compressed text indexing techniques [32], 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 Boldi and Vigna.

2.2 Rank and Select on Sequences

In this work we make use of compact data structures to manipulate sequences of symbols. In the simplest case we consider bitmaps (i.e., binary sequences) that are able to answer *rank* and *select* queries. *Rank* counts the number of 1s in a given prefix of the sequence and *select* finds the position of the i-th occurrence of a 1 in the bitmap.

There are many constant-time solutions for the *rank/select* problem on bitmaps $B[1, n]$. One of them requires $n + o(n)$ space (that is, $o(n)$ bits on top of B itself) [12, 28]. An improvement to this solution [34] retains constant-time queries while using $nH_0(B) + o(n)$ bits of space to represent B and the extra data structures. $H_0(B)$ corresponds to the zero-order entropy of bitmap B: The zero-order entropy for a binary sequence $B[1, n]$ with n_0 zeros and n_1 ones is

$$H_0(B) = \frac{n_0}{n} \log \frac{n}{n_0} + \frac{n_1}{n} \log \frac{n}{n_1} \ .$$

Rank and *select* operations can be extended to arbitrary sequences drawn from an alphabet Σ of size σ. The operations supported are: *access(i)* retrieves the character at position i; *rank(a, i)* counts the number of occurrences of a until position i; and *select(a, i)* returns the position where the i-th occurrence of the character a appears.

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

Golynski et al. [15] presented a data structure capable of performing these three operations in a sequence $S[1,n]$ using $n\log\sigma + o(n\log\sigma)$ bits and $O(\log\log\sigma)$ time. Note that $n\log\sigma$ is the space required by a plain representation of the sequence. Ferragina et al. [14] achieve zero-order compression, that is, $nH_0(S) + o(n\log\sigma)$ bits of space, and $O(1 + \frac{\log\sigma}{\log\log n})$ time per operation (this is a constant if $\sigma = O(\text{polylog}(n))$). The zero-order entropy formula generalizes to sequences as follows:

$$H_0(S) = \sum_{a\in\Sigma} \frac{n_a}{n} \log \frac{n}{n_a} \ ,$$

where n_a is the number of occurrences of symbol a in S.

The solution by Ferragina et al. builds over an elegant structure called the *wavelet tree* [17, 32]. This is a perfect binary tree where the root stores a bitmap formed by the n highest bits of each symbol in the sequence. Those symbols with highest bit 0 are then sent to the left subtree, and those with 1 to the right subtree. The decomposition continues recursively with the next highest bit, and so on. The tree has σ leaves and overall stores $n\log\sigma$ bits, just as the original sequence. If, however, those bitmaps are compressed to their zero-order entropy [34], the wavelet tree over the sequence $S[1,n]$ requires overall space $nH_0(S) + o(n\log\sigma)$. It implements *access*, *rank*, and *select* via $\log\sigma$ constant-time *rank*/*select* operations on the bitmaps. Ferragina et al. [14] improve upon this result by using multiary wavelet trees.

3 Re-Pair and Our Approximate Version

Re-Pair [26] 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. 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 $s \to ab$ to a dictionary R, where s is a new symbol not appearing in T.
3. It replaces every occurrence of ab in T by s.[4]
4. It iterates until every pair in T appears once.

Let us call C the resulting text (i.e., T after all the replacements). It is easy to expand any symbol s from C in time linear on the expanded data (that is, optimal): We expand s using rule $s \to s's''$ in R, and continue recursively with s' and s'', until we obtain the original symbols of T.

As each new rule added to R costs two integers of space, replacing pairs that appear twice does not involve any gain unless R is compressed. In the original proposal [26] a very space-effective dictionary compression method is presented. However, it requires R to be fully decompressed before using it. In this paper we are interested in being able to *operate* the graphs in little space. Thus, we favor a second technique to compress R [16], which reduces its space to about a half and can operate on the compressed representation. We use this dictionary representation in our experiments.

Despite its quadratic appearance, Re-Pair can be implemented in linear time [26]. However, this requires several data structures to track the pairs that must be replaced. This is usually

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

problematic when applying it to large sequences, as witnessed when using it for natural language text compression [39]. Indeed, it was also a problem when using it over suffix arrays [16], where a very successful approximate algorithm (that is, it does not always choose the most frequent pair to replace) was devised. The approximate algorithm runs very fast, with limited extra memory, and loses very little compression. Unfortunately, it only applies to suffix arrays.

We present now an alternative approximate Re-Pair compression 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.

3.1 Approximate Re-Pair

In this section we describe the method assuming we have $M > |T|$ main memory available, that is, the text fits in main memory. Section 3.2 considers the case of larger texts.

We place T inside the bigger array of size M, and use the remaining space as a (closed) hash table H of size $|H| = \min(M - |T|, 2|T|)$. Table H stores unique pairs of symbols $ab = t_i t_{i+1}$ occurring in T, and a counter of their number of occurrences in T. The key $ab = t_i t_{i+1}$ is represented as a single integer by its position i in T (any occurrence works). Thus each entry in H requires two integers.

The algorithm carries out several *passes*. At each pass, we identify the k most promising replacements to carry out, and then try to materialize them. Here $k \geq 1$ is a time/quality tradeoff parameter. At the end, the new text is shorter and the hash table can grow. We detail now the steps carried out for each pass.

Step 1 (counting pair frequencies). 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 $0 < \alpha < 1$ (defined by efficiency considerations), we do not insert new pairs anymore, yet we keep traversing T to increase the counters of already inserted pairs. This step requires $O(|T|) = O(n)$ time on average (the constant depends on α).

Step 2 (finding k promising pairs). We scan H and retain the k most frequent pairs from it. A heap of k pointers to cells in H is sufficient for this purpose. Hence we need also space for k further integers. This step requires $O(|H| \log k) = O(n \log k)$ time.

Step 3 (simultaneous replacement). The k pairs identified will be simultaneously replaced in a single 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). These considerations complicate the process.

We first empty H and reinsert only the k pairs to be replaced. This time we store the explicit key ab in the table, as well as a field *pos*, the position of its first occurrence in T. Special values for *pos* are *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 *null*, then this is its first occurrence, whose position we now record in *pos* (that is, we do not immediately replace the first occurrence until we are not sure

there will be at least two occurrences to replace). If, on the other hand, its *pos* value is *proceed*, we just replace ab by sz in T, where s is the new symbol for pair ab and z is an invalid entry. Finally, if pair ab already has a first position recorded in *pos*, we read this position in T and if it still contains ab (after possible replacements that occurred since 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 (i.e., its new first position). This method ensures that we create no new symbols s that will appear just once in T. It takes $O(|T|) = O(n)$ time on average.

Step 4 (compacting T and enlarging H). 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 $|H| = \min(M - |T|, 2|T|)$. The traversal of T, regarded as a circular array, will now start at the point where we stopped inserting pairs in H in Step 1 of the previous pass, to favor a uniform distribution of the replacements. This step takes $O(|T|) = O(n)$ time.

Approximate analysis. Although not being complete, the following analysis helps understand the accuracy/time tradeoff involved in the choice of k. Assume the exact method creates $|R|$ new symbols. The approximate method can also carry out $|R|$ replacements (achieving hopefully similar compression, since these need not be the same replacements of the exact method) in $p = \lceil |R|/k \rceil$ passes, which take overall average time $O(\lceil |R|/k \rceil \; n \log k)$. Thus we can trade time for accurateness by tuning k. The larger k, the faster the algorithm (as there is an $O(\log(k)/k)$ factor), but the less similar the result compared to the exact method. This analysis, however, is only an approximation, as some replacements could be invalidated by others and thus we cannot guarantee that we carry out k of them per round. Hence it could be that p is larger than $\lceil |R|/k \rceil$.

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 the algorithm becomes more accurate (thanks to a larger H) in its later stages, as by that time the frequency distribution is flatter and more precision is required to identify the best pairs to replace.

3.2 Running on Disk

The process described above also works well if T is too large to fit in main memory. In this case we maintain T on disk and table H occupies almost all the main memory, $|H| \approx M < |T|$. We must also reserve sufficient main memory for the heap of k elements. To avoid random accesses to T in Step 1, we do not store anymore in H the position of pairs ab, but instead ab explicitly. Thus Step 1 carries out a sequential traversal of T. Step 2 runs entirely in main memory. Step 4 involves another sequential traversal of T.

Step 3 is, again, the most complicated part. In principle, a sequential traversal of T is carried out. However, when a *pos* value changes to *proceed*, we make two replacements: one at its first occurrence (at value *pos*) and one at the current position in the traversal of T. The first involves a random access to T. Yet, this occurs only when we make the first replacement of an occurrence of a pair ab. This occurs at most k times per pass. However, checking that the first position *pos* still contains ab and has not been overwritten, involves another random access to T, and these cannot be bounded.

To carry out Step 3 efficiently, we note that there are at most k positions in T needing random access at any time, namely, those containing the *pos* ($\notin \{null, proceed\}$) values of the k pairs to

be replaced. We maintain those k disk pages cached in main memory. Those must be replaced whenever value *pos* changes. This replacement does not involve reading a new page, because the new *pos* value always corresponds to the current traversal position (whose block is also cached in main memory). Thus cached pages not pointed anymore from any *pos* values are simply discarded (hence an elementary reference counting mechanism is necessary), and the current page of T might be retained in main memory if, after processing it, some *pos* values now point to it.

As explained, most changes to T are done at the current traversal position, hence it is sufficient to write back the current page of T after processing it to handle those changes. The exceptions are the cases when one writes at some old position *pos*. In those cases the pages we have cached in main memory must be written back to disk. Yet, as explained, this occurs at most k times per pass. (Note that using a dirty bit for the cached pages might avoid some of those write-backs, as the dirty page could be modified several times before being abandoned by all the pairs.)

Thus the worst-case I/O cost of this algorithm, if p passes are carried out, is $O(p\ (n/B + k))$, where B is the disk block size. That is, the algorithm is almost I/O optimal with respect to its main memory version. Indeed, it is asymptotically I/O optimal if $k = O(n/B)$, which is a rather reasonable condition.

4 A Compressed Graph Representation using Re-Pair

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. Figure 1 shows an example graph, and Figure 2 illustrates part of the execution of our approximate Re-Pair algorithm over that graph.

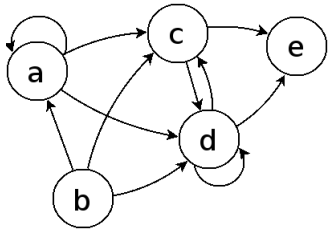

Figure 1: An example graph.

The application of Re-Pair to $T(G)$ has several important properties:

- 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. Moreover, Re-Pair handles well large alphabets, V in our case.

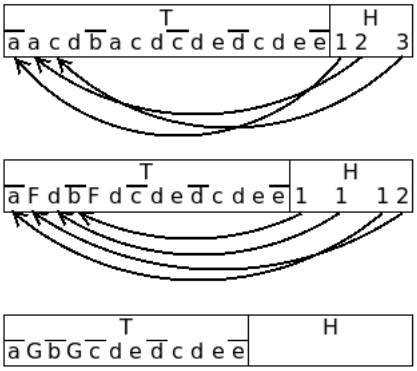

Figure 2: Part of the execution of the approximate version of Re-Pair over the graph of Figure 2. H represents the space used for the hash table and T the space used for the text. The arrows point from the counter in the hash table to the first occurrence of the pair counted by that field. In the first iteration we replace ac by F; we do not replace cd because ac blocks every replacement. During the second iteration we replace Fd by G.

- 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. This can be enforced at compression time, at the expense of losing some compression ratio, by preempting the compression algorithm when $|R|$ reaches the memory limit.

- 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 [16] 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 by Boldi and Vigna [8] 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 Boldi and Vigna [8] achieve by explicitly encoding the differences using gaps, 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 Boldi and Vigna's

work [8] 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 remain unaltered in C). Using essentially 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. Integers in C are stored using the minimum bits required to store the maximum value in C.

4.1 Improvements

We describe now several possible improvements over the basic scheme. Some can be combined, some not. Several possible combinations are explored in the experiments.

Differential encoding. 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 encode every adjacency list 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 many nodes tend 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. The price is slightly slower decompression.

Reordering lists. Since the adjacency list does not need to be output in any particular order, we can alter the original order to spot more global similarities. Consider the lists $1,2,3,4,5$ and $1,2,4,5$. Re-Pair can replace $1,2$ by 6 and $4,5$ by 7, but the common subsequence $1,2,4,5$ cannot be fully exploited because the first list has a 3 in between. If we sort both adjacency lists after compressing we get $3,6,7$ and $6,7$, and then we can replace $6,7$, thus exploiting global regularities in both adjacency lists. The method is likely to improve compression ratios. The compression process is slightly slower: it works almost as in the original version, except that the lists are sorted after each pass of Re-Pair. Decompression and traversal, on the other hand, are not affected at all. Note that it is not possible to combine this method with differential encoding.

Removing pointers. It might be advantageous, for relatively sparse graphs, to remove the need to spend a pointer for each node (to the beginning of its adjacency list in C). We can replace the pointers by two bitmaps. The first one, $B_1[1,n]$, marks in $B_1[i]$ whether node v_i has a non-empty adjacency list. The second bitmap, $B_2[1,c]$ (where $c = |C| \leq m$), marks the positions in C where adjacency lists begin. Hence the starting position of the list for node v_i in C is $select(B_2, rank(B_1, i))$ if $B_1[i] = 1$ (otherwise the list is empty). The list extends up to the next 1 in B_2. The space is $n + c + o(n + c)$ bits, instead of $n \log c$ needed by the pointers. When n is significant compared to c, space reduction is achieved at the expense of slower access to the adjacency lists.

5 Lempel-Ziv Compression of Web Graphs

The Lempel-Ziv compression family [40, 41] achieves compression by replacing repeated sequences found in the text by a pointer to a previous occurrence thereof. In particular, the LZ78 variant [41] stands as a plausible alternative candidate to Re-Pair for our goals: it detects duplicate lists of links in the adjacency lists, handles well large alphabets, and permits fast local decompression. Moreover, LZ78 admits efficient compression without requiring approximations.

5.1 The LZ78 Compression Algorithm

LZ78 compresses the text by dividing it into *phrases*. Each phrase is built as the concatenation of the longest previous phrase that matches the prefix of the text yet to be compressed and an extra character which makes this phrase different from all the previous ones. The algorithm is as follows:

1. It starts with a *dictionary* S of known phrases, containing initially the empty string.
2. It finds the longest prefix $T_{i,j}$ of the text $T_{i,n}$ yet to be processed, which matches an existing phrase. Let p be that phrase number.
3. It adds a new phrase to S, with a fresh identifier, and content (p, t_{j+1}).
4. It returns to step 2, to process the rest of the text $T_{j+2,n}$.

In order to carry out Step 2 efficiently, S is organized as a trie data structure. The output of the compressor is just the sequence of pairs (p, t_{j+1}). The phrase identifier is implicitly given by the position of the pair in the sequence.

The text of any phrase in the compressed text can be obtained backwards in optimal time. Let p_0 the phrase we wish to expand. We read the p_0-th pair in the compressed sequence and get (p_1, c_0). Then c_0 is the last character of the phrase. Now we read the p_1-th pair and get (p_2, c_1), thus c_1 precedes c_0. We continue until reaching $p_i = 0$, which denotes the empty phrase. In i constant-time steps we obtained the content $c_{i-1}c_{i-2}\ldots c_1c_0$.

Just as Re-Pair, this extraction can be made I/O-optimal if we limit the creation of phrases to what can be maintained in main memory. After that point, the process continues identically but no new phrases are inserted into S (hence not all the phrase contents will be different).

5.2 Using LZ78 for Graph Compression

For a graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $adj(v_i) = \{v_{i1}, v_{i2}, \ldots, v_{ia_i}\}$ is the set of neighbors of node v_i, the textual representation used for LZ78 compression is slightly different from that of Section 4:

$$T = T'(G) = v_{11}v_{12}v_{13}\ldots v_{1a_1}v_{21}v_{22}\ldots v_{2a_2}\ldots v_{n1}v_{n2}\ldots v_{na_n},$$

where we note that the special symbols $\overline{v_i}$ have been removed. The reason is that removing them later is not as easy as for Re-Pair. To ensure that adjacency lists span an integral number of phrases (and therefore can be extracted in optimal time $O(|adj(v_i)|)$), we run a variant of LZ78 compression. In this variant, when we look for the longest phrase $T_{i,j}$ in Step 2, we never cross a list boundary. More precisely, the character t_{j+1} to be appended to the new phrase must still

belong to the current adjacency list. This might produce repeated phrases in the compressed text, which of course are not inserted into S.

Like C, the array of pointers and symbols added are stored using the minimum number of bits required by the largest pointer and symbol, respectively.

In addition, we store a pointer to every beginning of an adjacency list in the compressed sequence, just as for Re-Pair. Some of the improvements in Section 4.1 can be applied as well: differential encoding (which will have a huge impact with LZ78) and replacing pointers by bitmaps.

6 Experimental Results

We carried out several experiments to measure the compression and time performance of our graph compression techniques, comparing them to the state of the art.

We downloaded four Web crawls from the WebGraph project, `http://law.dsi.unimi.it/`. Table 1 shows their main characteristics. The last column shows the size required by a plain adjacency list representation using 4-byte integers.

Crawl	Nodes	Edges	Edges/Nodes	Plain size (MB)
`EU`	862,664	19,235,140	22.30	77
`Indochina`	7,414,866	194,109,311	26.18	769
`UK`	18,520,486	298,113,762	16.10	1,208
`Arabic`	22,744,080	639,999,458	28.14	2,528

Table 1: Some characteristics of the fours crawls used in our experiments.

6.1 Compression Performance

Our compression algorithm is parameterized by M, k, and α. Those parameters yield a tradeoff between compression time and compression effectiveness. In this section we study those tradeoffs. As there are several possible variants of our method, we stick in this section to the one called *Re-Pair Diffs CDict NoPtrs* in Section 6.3. The machine used in this section is a 2GHz Intel Xeon (8 cores) with 16 GB RAM and 580 GB Disk (SATA 7200rpm), running Ubuntu GNU/Linux with kernel 2.6.22-14 SMP (64 bits). The code was compiled with `g++` using the `-Wall`, `-O9` and `-m32` options. The space is measured in bits per edge (bpe), dividing the total space of the structure by the number of edges in the graph.

Parameter α (the maximum loading ratio of the hash table H before we stop inserting new pairs) turns out to be not too relevant, as its influence on the results is negligible for a wide range of reasonable choices. We set $\alpha = 0.6$ for all our experiments.

Value M is related to the amount of extra memory we require on top of T. Our first experiment aims at demonstrating that we obtain competitive results using very little extra memory. Table 2 shows the compression ratios achieved with different values of M (as a percentage over the size of T). As it can be seen, we gain little compression by using more than 5% over $|T|$, which is extremely modest (the linear-time exact Re-Pair algorithm [26] uses at the very least 200% extra space). The rest of our experiments are run using 3% extra space[5].

[5]That is, in the beginning. As the text is shortened along the compression process we enlarge the hash table and

Graph	1%	3%	5%	10%	50%
EU	4.68	4.47	4.47	4.47	4.47
Indochina	2.53	2.53	2.53	2.52	2.52
UK	4.23	4.23	4.23	4.23	4.23
Arabic	3.16	3.16	3.16	3.16	3.16

Table 2: Compression ratios (in bpe) achieved when using different amounts of extra memory for H (measured in percentage over the size of the sequence to compress). In all cases we use $k = 10,000$.

We now study the effect of parameter k in our time/quality compression tradeoff. Table 3 shows the time and compression ratio achieved for different k on our crawls. For the smaller crawls we also run the exact algorithm (using a relatively compact implementation [16] that requires 260MB total space for EU and 2.4GB for Indochina). It can be seen that our approximate method is able of getting very close to the exact result while achieving reasonable performance (around 1 MB/sec). Lempel-Ziv compression is much faster but compresses far less.

It is interesting to notice that, as k doubles, compression time is almost halved (especially for small k). This is related to the approximate analysis of our methods, where we could not guarantee that all the k pairs chosen are actually replaced. Table 4 measures the number of replacements actually done by our algorithm on crawls EU and Indochina. As it can be seen, for k up to 10,000, more than 85% of the planned replacements are actually carried out, and this improves for larger graphs. Note also that the number of passes made by the algorithm is rather reasonable. This is relevant for secondary memory, as it means for example that with $k = 10,000$ we expect to do about 60 passes over the (progressively shrinking) text on disk for the EU crawl, and 263 for the Indochina crawl.

For the rest of the experiments we use $k = 10000$.

6.2 Limiting the Dictionary

As explained, we can preempt Re-Pair compression at any pass in order to limit the size of the dictionary. This is especially interesting when the graph, even in compressed form, does not fit in main memory. In this case, we can take advantage of the locality of accesses to C to speed up the access to the graph: If we are able of compressing $T(G)$ by a factor c, then access to long adjacency lists can be speeded up by a factor up to c. However, some Re-Pair structures need random access, and those must reside in RAM. This includes the dictionary, but also the structure that tells us where each adjacency list starts in C. The latter could still be kept on disk at the cost of one extra disk access per list, whereas the former definitely needs to lie in main memory.

Figure 3 shows the tradeoffs achieved between the size of the main sequence C and that of the RAM structures, as we modify the preemption point. It is interesting to notice that the main memory usage has a minimum, due to the fact that, as compression progresses, the dictionary grows but the width of the pointers to C decreases[6].

At those optima, the overall size of C plus RAM data is not the best possible one, but rather close. In our graphs, the optimum space in RAM is from 0.2 to 0.4 bpe. This means, for example,

keep using the absolute space originally allowed.

[6]In the variant *NoPtrs* we use a bitmap of $|C|$ bits, which produces the same effect.

`EU`

k	time (min)	bpe
exact	86.15	4.40
10,000	1.77	4.47
25,000	1.03	4.70
50,000	0.83	4.74
75,000	0.72	4.76
100,000	0.73	4.79
250,000	0.62	4.91
500,000	0.62	4.95
1,000,000	0.67	4.95
LZ Diffs	0.07	7.38

`Indochina`

k	time (min)	bpe
exact	5,230.67	2.50
10,000	52.97	2.53
25,000	20.73	2.53
50,000	12.68	2.54
75,000	8.70	2.54
100,000	7.75	2.54
250,000	4.85	2.56
500,000	4.07	2.59
1,000,000	3.77	2.62
LZ Diffs	0.53	4.89

`UK`

k	time (min)	bpe
10,000	341.32	4.23
25,000	142.57	4.24
50,000	74.20	4.25
75,000	49.08	4.25
100,000	38.22	4.25
250,000	20.45	4.26
500,000	14.23	4.27
1,000,000	10.60	4.29
LZ Diffs	1.32	8.56

`Arabic`

k	time (min)	bpe
10,000	1,034.53	3.16
25,000	370.08	3.18
50,000	191.60	3.19
75,000	132.72	3.19
100,000	102.55	3.19
250,000	53.77	3.20
500,000	30.48	3.21
1,000,000	24.57	3.23
LZ Diffs	2.72	6.11

Table 3: Time for compressing different crawls with different k values. For the smaller graphs we also include the exact method. We also include the results of our LZ variants for the four crawls. The LZ version was compiled without the `-m32` flag, since our implementation requires more than 4GB of RAM for the larger graphs.

that just 15MB of RAM is needed for our largest graph, `Arabic`. If we extrapolate to the 600GB graph of the *whole* static indexable Web, we get that we could handle it in secondary memory with a commodity desktop machine of 4GB to 8GB of RAM. If the compression would stay at about 6 bpe, this would mean that access to the compressed Web graph would be up to 5 times faster than in uncompressed form, on disk.

6.3 Compressed Graphs Size and Access Time

We now study the space versus access time tradeoffs of our graph compression proposals based on Re-Pair and LZ78. From all the possible combinations of improvements[7] depicted in Sections 4 and 5 we have chosen the following, which should be sufficient to illustrate what can be achieved (see in particular Section 4.1).

[7]We can devise 16 combinations of Re-Pair and 8 combinations of LZ78 variants.

EU

k	Passes	Total Pairs	Pairs/pass	% of k
5,000	108	497,297	4,604	92.08
10,000	58	502,530	8,664	86.64
20,000	33	513,792	15,569	77.85
50,000	19	543,417	28,600	57.20
100,000	14	576,706	41,193	41.19
500,000	12	676,594	56,382	11.28
1,000,000	12	676,594	56,382	5.64

Indochina

k	Passes	Total Pairs	Pairs/pass	% of k
10,000	263	2,502,880	9,516	95.16
20,000	136	2,502,845	18,403	92.02
50,000	60	2,503,509	41,725	83.45
100,000	34	2,528,530	74,368	74.37
500,000	16	2,772,091	173,255	34.65
1,000,000	14	2,994,149	213,867	21.39
5,000,000	14	3,240,351	231,453	4.63
10,000,000	14	3,240,351	231,453	2.31

Table 4: Number of pairs created by approximate Re-Pair over two crawls.

- *Re-Pair*: Normal Re-Pair.
- *Re-Pair Diffs*: Re-Pair with differential encoding.
- *Re-Pair Diffs NoPtrs*: Re-Pair with differential encoding and with pointers to C replaced by bitmaps.
- *Re-Pair Diffs CDict NoPtrs*: Re-Pair with differential encoding and a compacted dictionary. In the other implementations, every element of the dictionary is stored as an integer in order to speed up the access. This version stores every value using the required number of bits and not 32 by default. It also replaces the pointers to C by bitmaps.
- *Re-Pair Reord*: Normal Re-Pair with list reordering.
- *Re-Pair Reord CDict*: Re-Pair with list reordering and compacted dictionary.
- *LZ*: Normal LZ78.
- *LZ Diffs*: LZ78 on differential encoding.

For each of those variants, we measured the size needed by the structure versus the time required to access random adjacency lists. Structures that offer a space/time tradeoff will appear as a line in

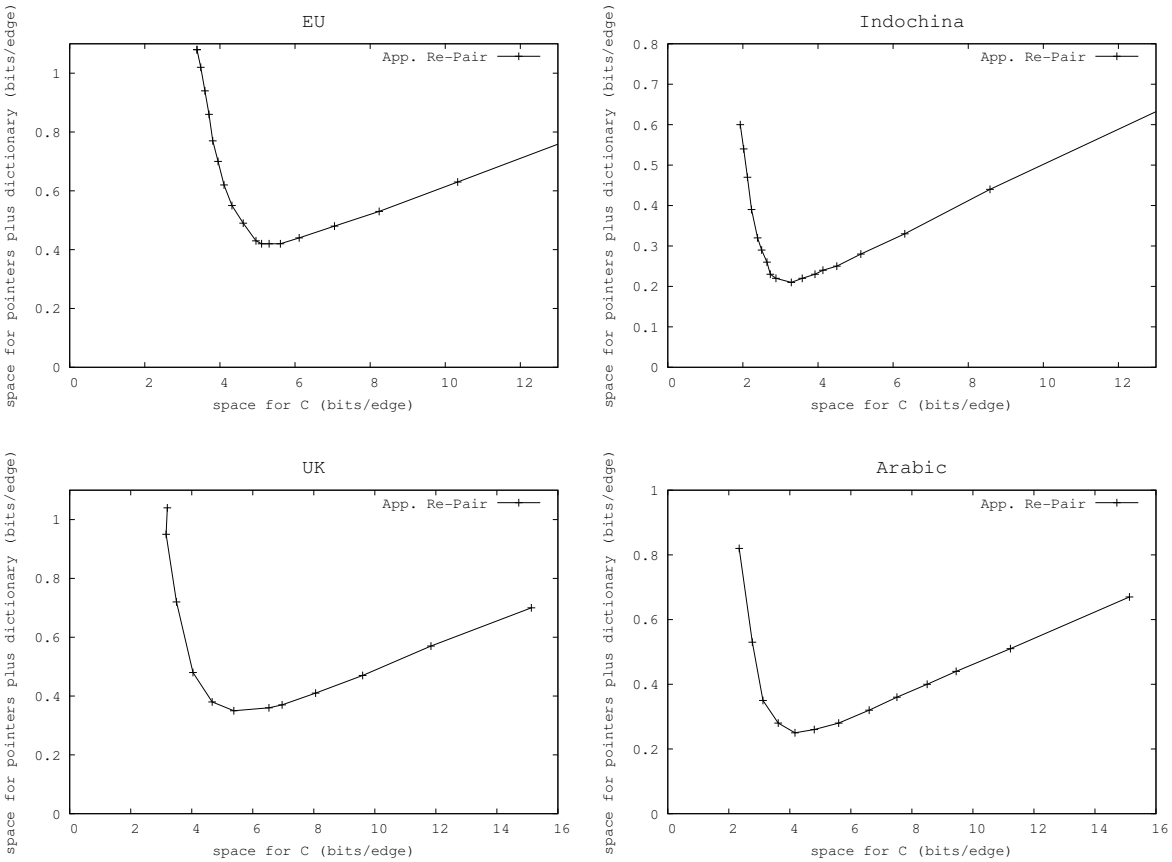

Figure 3: Space used by the sequence versus the dictionary plus the pointers, all measured in bits per edge.

this plot, otherwise they will appear as points. The time is measured by extracting full adjacency lists and then computing the time per extracted element in $adj(v_i)$. More precisely, we generate a random permutation of all the nodes in the graph and sum the user time of recovering all the adjacency lists (in random order). The time per edge is this total time divided by the number of edges in the graph.

These 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.

We compared to Boldi & Vigna's implementation [8] run on our machine with different space/time tradeoffs. The implementation of Boldi & Vigna gives a size measure that is consistent with the sizes of the generated files (and with their paper [8]). However, their process (in Java) needs significantly more memory to run. This could suggest that they actually use some structures that are not stored on the file, but built on the fly at loading time. Those should be accounted for in order to measure the size needed by the data structure to operate. Yet, 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++, the Java compiler achieves very competitive results[8].

We also show, in a second plot, a comparison of our variants with plain adjacency list representations. One representation, called "plain", uses 32-bit integers for nodes and pointers. A second one, called "compact", uses $\lceil \log_2 n \rceil$ bits for node identifiers and $\lceil \log_2 m \rceil$ for pointers to the adjacency list.

Figures 4 to 7 show the results for the four Web crawls. The different variants of LZ achieve the worst compression ratios (particularly without differences), but they are the fastest (albeit for a very little margin). The normal Re-Pair achieves a competitive result both in time and space. The other variants achieve different competitive space/time tradeoffs. The most space-efficient variant is *Re-Pair Diffs CDict NoPtrs*.

Node reordering usually achieves better compression without any time penalty, yet it cannot be combined with differential encoding.

A similar time/space tradeoff shown between *Re-Pair Diffs* and *Re-Pair Diffs NoPtrs* can be achieved with the other representations that use Re-Pair, since the pointers are the same for all of them. The time/space tradeoff between compacting the dictionary or not should be almost the same for the other Re-Pair implementations too.

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 can achieve almost the same space (and less in some cases). Moreover, using the same amount of space, our method is always faster (usually 2–3 times faster, even considering their best line). In addition, some of our versions (those that do not use differential encoding) do not impose any particular node numbering.

Compared to an uncompressed graph representation, our method is also a very interesting alternative. It is 3–10 times smaller than the compact version and 2–4 times slower than it; and it is 5–13 times smaller than the the plain version and 4–8 times slower.

7 Conclusions and Future Work

We have presented a graph compression method that exploits the similarities between adjacency lists by using grammar-based compressors such as Re-Pair [26] and LZ78 [41]. Our results demonstrate that those similarities account for most of the compressibility of Web graphs, on which our technique performs particularly well. Our experiments over different Web crawls demonstrate that our method achieves compression ratios very close to (and sometimes slightly better than) those of the best current schemes [8], while being 2–3 times faster to navigate the compressed graph. Compared to a plain adjacency list representation, our compressed graphs can be 5 to 13 times smaller, at the price of a 4- to 8-fold traversal slowdown (this has to be compared to the hundred to thousand times slowdown caused by running on secondary memory). This makes it a very attractive choice to maintain graphs all the time in compressed form, without the need of a full decompression in order to access them. As a result, graph algorithms that are designed for main memory can be run over much larger graphs, by maintaining them in compressed form. In cases the graphs do not fit

[8] See `http://www.idiom.com/~zilla/Computer/javaCbenchmark.html` or `http://www.osnews.com/story/5602`.

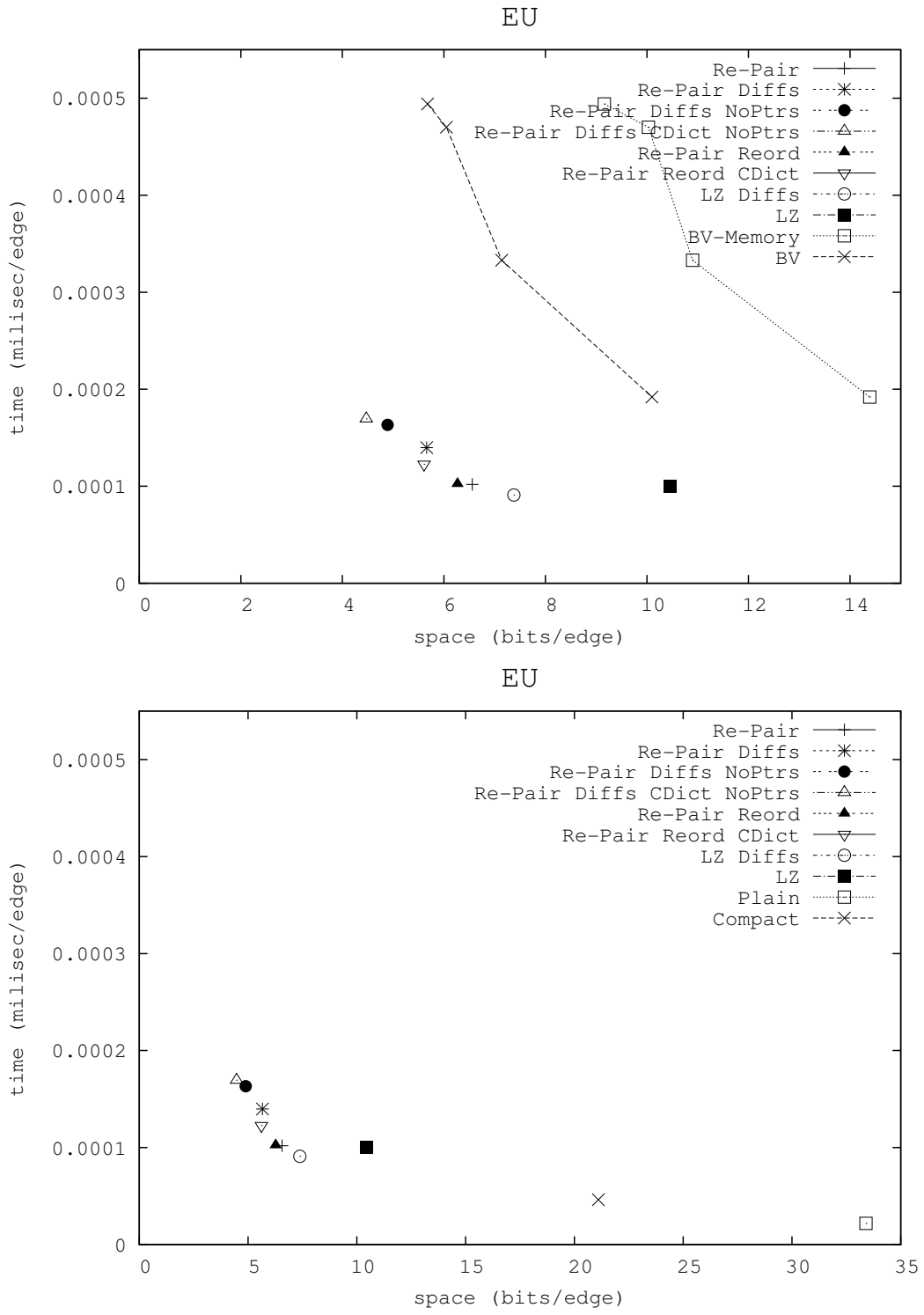

Figure 4: Space and time to find neighbors for different graph representations, over `EU` crawl. BV-Memory represents the minimum heap space needed by the process to run.

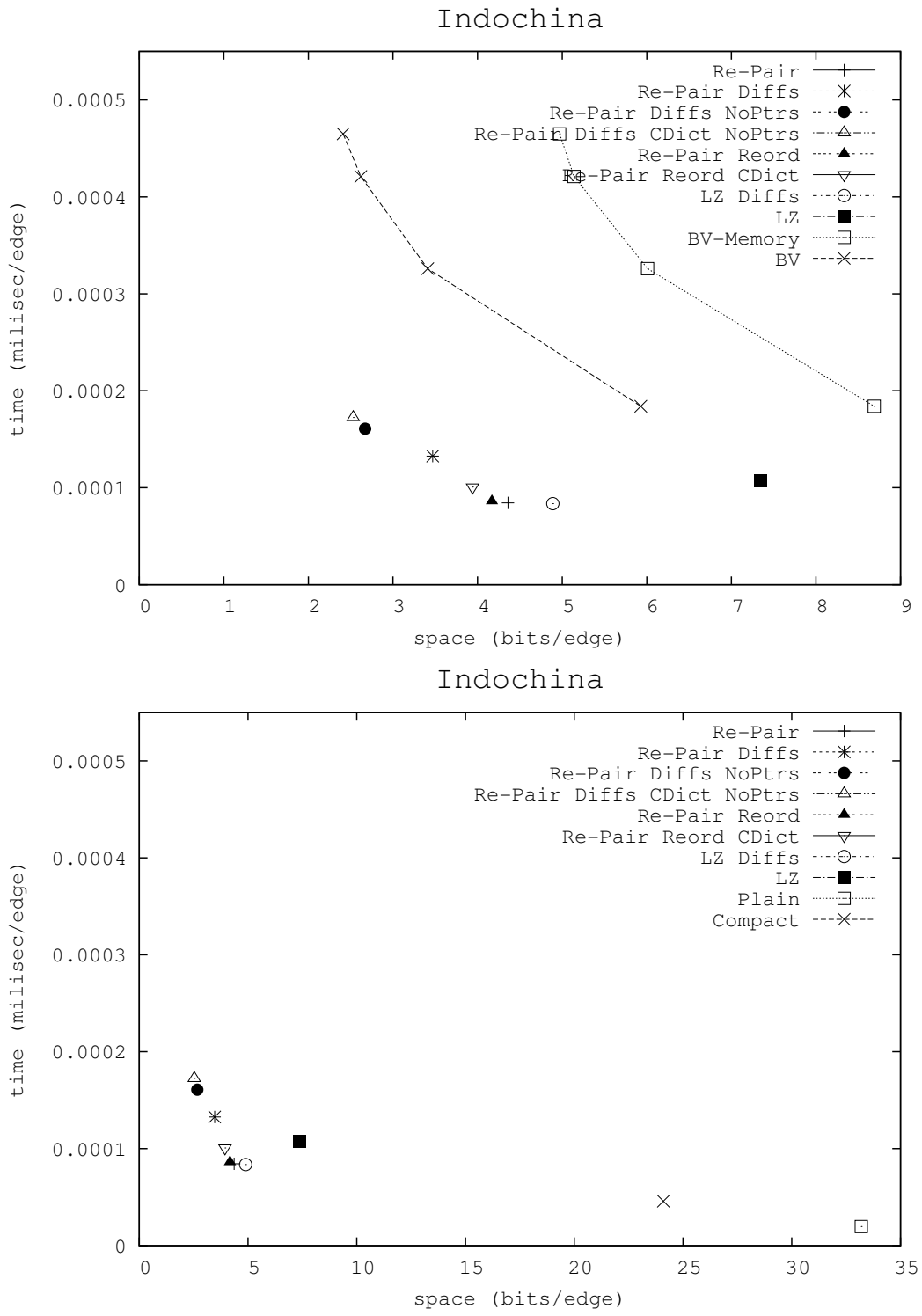

Figure 5: Space and time to find neighbors for different graph representations, over `Indochina` crawl. BV-Memory represents the minimum heap space needed by the process to run.

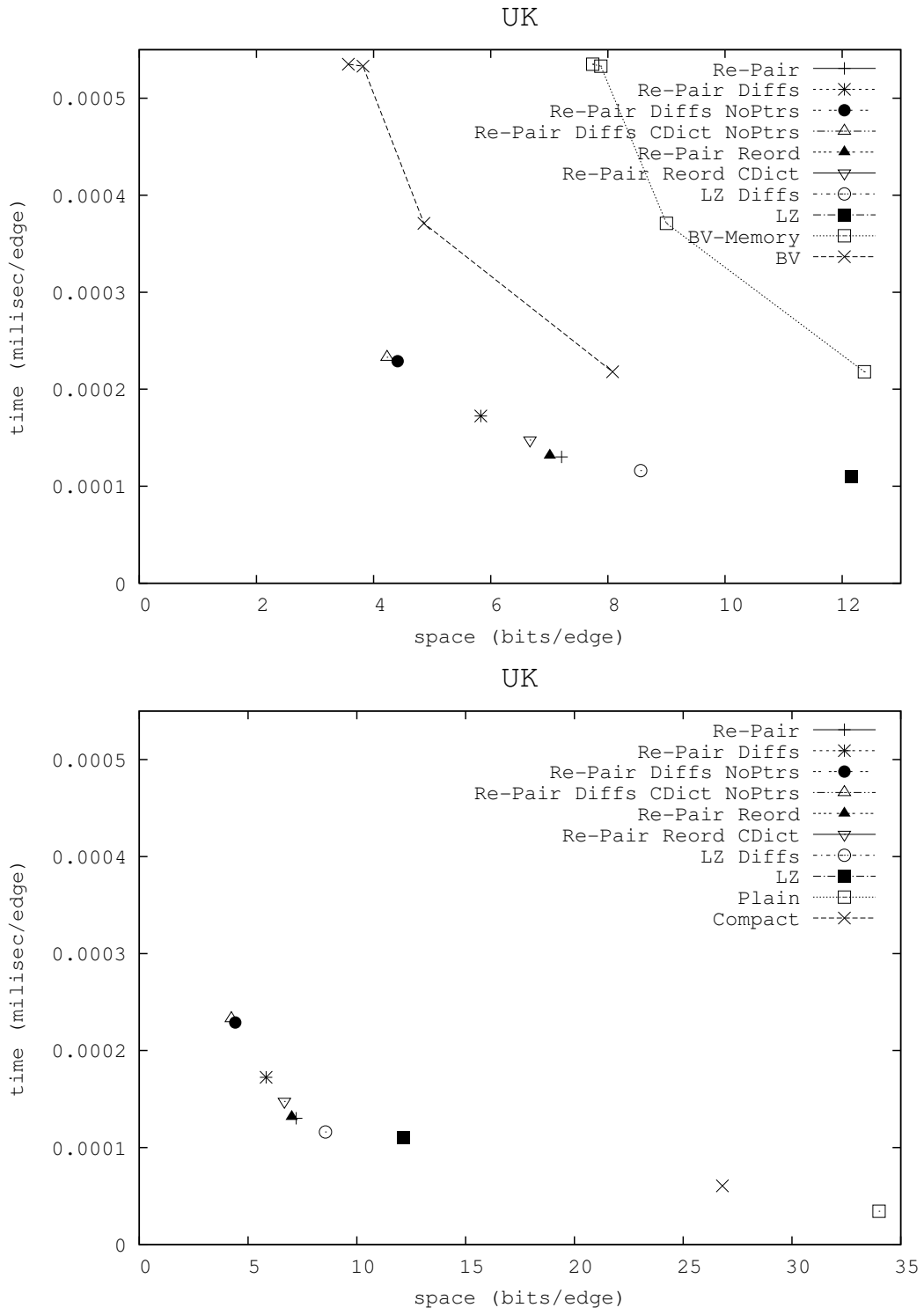

Figure 6: Space and time to find neighbors for different graph representations, over UK crawl. BV-Memory represents the minimum heap space needed by the process to run.

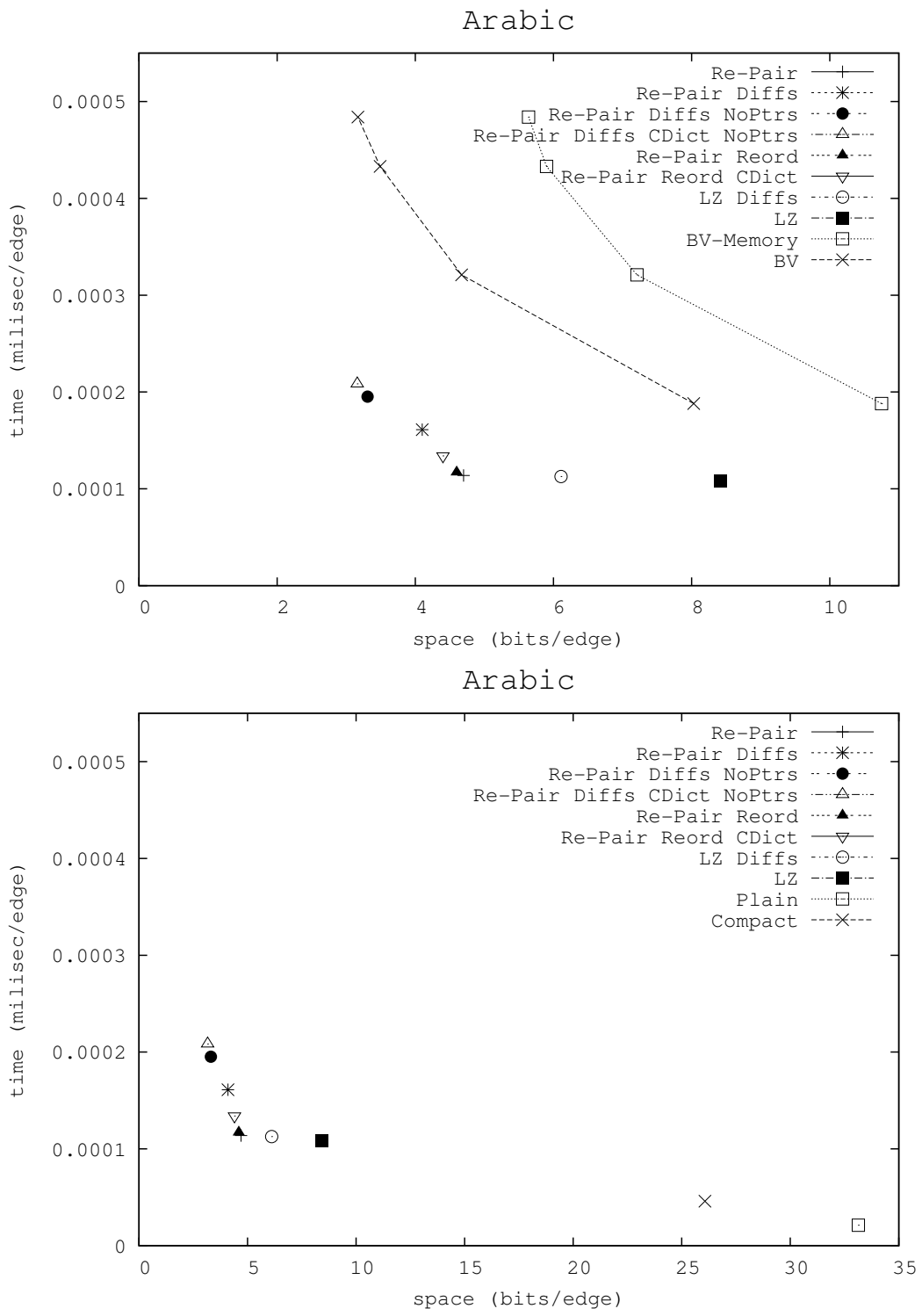

Figure 7: Space and time to find neighbors for different graph representations, over `Arabic` crawl. BV-Memory represents the minimum heap space needed by the process to run.

in main memory even in compressed form, our scheme adapts well to secondary memory, where it can take fewer accesses to disk than its uncompressed counterpart for navigation.

As a byproduct, we developed an efficient approximate version of Re-Pair, which can work within very limited space and also works well on secondary memory. This can be of independent interest given the large amount of memory required by the exact Re-Pair compression algorithm.

Finally, our technique is not particularly tailored to Web graphs (more than trying to exploit similarities in adjacency lists). This could make it suitable to compress other types of graphs, whereas other approaches which are too tailored to Web graphs could fail. To us, this is a beautiful example where a general and elegant technique can compete successfully with carefully ad-hoc designed schemes. As we all know, this type of fortunate situations do not arise too frequently.

There are still several interesting lines for future work, which we briefly discuss next.

7.1 Further Compression

The compressed sequence C is still stored with fixed-length integers. This is amenable of further compression: After applying Re-Pair, every pair of symbols in C is unique, yet individual symbols are not. Thus, zero-order compression could still reduce the space (probably at the expense of increasing the access time).

Yet, it is not immediate how to apply a zero-order compressor to such sequence, because its alphabet is very large. For example, applying Huffman would be impractical because of the need to store the table (i.e., at least the symbol permutation in decreasing frequency order). Instead, one could consider approximations such as Hu-Tucker's [21], which does not permute the symbols and thus needs only to store the tree shape. Hu-Tucker achieves less than 2 bits over the entropy.

To get a rough idea of what could be achieved, we estimated the space needed by Huffman and Hu-Tucker methods on our graphs, for the version *Re-Pair Diffs*. Let us call Σ the alphabet of C, and σ its size ($n \leq \sigma \leq n + |R|$), and say that n_i is the number of occurrences of the symbol i in C. We lower bound the maximum size that Huffman can achieve as:

$$\mathit{Huffman} \quad \geq \quad \sigma \log \sigma + \sum_{i \in \Sigma} n_i \log \frac{n}{n_i},$$

where we have optimistically bounded its output with the zero-order entropy and also assumed that the tree shape information is free (it is indeed almost free when using canonical Huffman codes, and the entropy estimation is at most 1 bit per symbol off, so the lower bound is rather tight).

Since Hu-Tucker achieves more competitive results, we lower and upper bound its performance:

$$2\sigma + \sum_{i \in \Sigma} n_i \log \frac{n}{n_i} \quad \leq \quad HT \quad \leq \quad 2\sigma + \sum_{i \in \Sigma} n_i \left(\log \frac{n}{n_i} + 2 \right),$$

where the term 2σ arises because we have to represent an arbitrary binary tree of σ leaves, so the tree has $2\sigma - 1$ nodes and we need basically $2\sigma - 1$ bits to represent it (e.g., using 1 for internal nodes and 0 for leaves).

Table 5 shows the compresion ratio bounds for C (i.e., not considering the other structures). As expected, Huffman compression is not promising, because just storing the symbol permutation offsets any possible gains. Yet, Hu-Tucker stands out as a promising alternative to achieve further

compression. However, because of the bit-wise output of these zero-order compressors, the pointers to C must be wider[9]. Table 6 measures the size of the whole data structure with and without Hu-Tucker (we use the lower bound estimation for the latter). It can be seen that compression is not attractive at all, and in addition we will suffer from increased access time due to bit manipulations.

Graph	Huffman lower bound	Hu-Tucker lower bound	Hu-Tucker upper bound
`EU`	145.68%	84.65%	94.18%
`Indochina`	161.57%	82.11%	90.44%
`UK`	168.87%	82.94%	90.64%
`Arabic`	162.96%	82.81%	90.51%

Table 5: Compression ratio bounds for C, using *Re-Pair Diffs*. We measure the compressed C size as a percentage of the uncompressed C size.

Graph	Hu-Tucker (*Diff NoPtrs*)	Hu-Tucker (*Diff*)	Original
`EU`	6.61	4.89	4.47
`Indochina`	3.64	3.13	2.53
`UK`	6.14	5.33	4.23
`Arabic`	4.01	3.14	3.16

Table 6: Total space required by our original structures and the result after applying Hu-Tucker (lower-bound estimation).

An alternative, more sophisticated, approach to achieve zero-order entropy is to represent C using a wavelet tree where the bitmaps are compressed using the technique described in Section 2.2. This guarantees zero-order entropy (plus some sublinear terms for accessing the sequence), and it can take even less because each small chunk of around 16 entries of C is compressed to its own zero-order entropy. The sum of those zero-order entropies add up to at most the zero-order entropy of the whole sequence, but it can be significantly less if there are local biases of symbols (as it could perfectly be the case in Web graphs due to local references).

Our implementation of those wavelet trees uses a sampling method that permits accessing the compressed sequences at arbitrary points. The sparser the sampling, the slower the access but the lower the space. Table 7 shows some results on the achievable space. We note that, because we can still refer to entry offsets (and not bit offsets) in C, our pointers to C do not need to change (nor the *NoPtrs* bitmap). We achieve impressive space reductions, to 70%–75% of the original space, and for `Indochina` we largely break the 2 bpe barrier.

In exchange, symbol extraction from C becomes rather slow. We measured the access time per link for the `Arabic` crawl using a sample of 32, and found that this approach is 22 times slower than our smallest (and slowest) version based on Re-Pair. For a samplig of 128 the slowdown is 43.

This can be alleviated by extracting all the symbols from an adjacency list at once, as no new *rank* operations are needed once we go through the same wavelet tree node again. In the worst

[9]In the *NoPtrs* case this is worse, as we now need to spend one extra bit per *bit* of C, not per *number* in C.

case, we pay $O(k(1+\log\frac{\sigma}{k}))$ time, instead of $O(k\log\sigma)$, to extract k symbols. This improvement can only be applied when the symbols can be retrieved in any order, so it could not be combined with differences.

Our goal in this experiment was to show that it is still possible to achieve better results in terms of space, whereas more research is needed to couple those with attractive access times.

Graph	WT (8)	WT (32)	WT (128)	WT (∞)	Original
EU	4.59	3.71	3.49	3.42	4.47
Indochina	2.52	1.97	1.84	1.79	2.53
UK	4.36	3.40	3.16	3.08	4.23
Arabic	3.34	2.60	2.42	2.36	3.16

Table 7: Total space, measured in bpe, achieved when using compressed wavelet trees to represent C, with different sampling rates.

7.2 Extended Functionality

Retrieving the list of neighbors is just the most basic graph traversal operation. One could explore other relevant operations to support within compressed space. An obvious one is to know the indegree/outdegree of a node. Those can be stored in plain form using $O(n\log m)$ bits, or using bitmaps: if we write in a bitmap a 1 for each new adjacency list and a 0 for each new element in the current adjacency list, one can compute outdegree as $select(i+1)-select(i)$. This bitmap requires $m+n+o(m+n)$ bits of space (i.e., little more than 1 bpe on typical Web graphs). This could be compressed to $O(n\log\frac{m}{n})$ bits (around 0.5 extra bpe in practice on typical Web graphs) and still support *select* in constant time. Indegree can be stored with a similar bitmap.

More ambitious is to consider *reverse neighbors* (list the nodes that point to v_i), which permits backward traversal in the graph. One way to address this is to consider the graph as a *binary relation* on $V\times V$, and then use the techniques of Barbay et al. [3] for binary relations, where forward and reverse traversal operations can be solved in time $O(\log\log n)$ per node delivered. A more recent followup [4] retains those times and reduces the space to the zero-order entropy *of the binary relation*, that is, $\log\binom{m}{n}$.

This compression result is still poor for Web graphs, see Table 8 where we give lower bound estimations of the space that can be achieved (that is, we do not charge for any sublinear space terms on top of the binary relation data structures, which are significant in practice). The space for the binary relation is not much smaller than that of a plain adjacency list representation, although within that space it can solve reverse traversal queries. To achieve the same functionality we would need to store the original and the transposed graphs, hence we included the column "2 × Plain" in the table. We also included our best result based on Re-Pair. Even doubling that to support reverse queries would be much better than binary relations[10].

Indeed, we can regard our graph compression method as (and attribute its success to) the decompostion of the graph binary relation into two binary relations:

- Nodes are related to the Re-Pair symbols that conform their (compressed) adjacency list.

[10]The transposed graph could compress differently, but usually it compresses even better [8].

Graph	Plain	2 × Plain	Bin.Rel.	Our Re-Pair
EU	20.81	41.62	15.25	2.53
Indochina	23.73	47.46	17.95	3.16
UK	25.89	51.78	20.13	4.47
Arabic	25.51	51.02	19.66	4.23

Table 8: Expected space usage (bpe) using the binary relation method, compared to other results.

- Re-Pair symbols are related to the graph nodes they expand to.

Our result in this paper can be restated as: the graph binary relation can be efficiently decomposed into the product of the two relations above, achieving significant space gains. The regularities exposed by such a factorization go well beyond those captured by the zero-order entropy of the original binary relation. Now, representing these two binary relations with the technique of Barbay et al. would yield space comparable to our current solution, and $O(\log \log n)$ complexities to retrieve forward and reverse neighbors. The real slowdown that this solution would involve can only be measured with an implementation of that data structure, which does not yet exist as far as we know. Also, if we use this direct solution we would lose the compression we achieved in the dictionary.

Despite that our decomposition is tailored to Re-Pair compression, we believe that the perspective of achieving compressible decompositions of binary relations can be a very interesting research track on its own.

Acknowledgments

We thank Rodrigo Paredes for pointing out that reordering the adjacency lists would allow us to exploit more regularities.

References

[1] M. Adler and M. Mitzenmacher. Towards compressing Web graphs. In *Proc. 11th IEEE Data Compression Conference (DCC)*, pages 203–212, 2001.

[2] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proc. 32th ACM Symposium on Theory of Computing (STOC)*, pages 171–180, 2000.

[3] J. Barbay, A. Golynski, I. Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proc. 17th Symposium on Combinatorial Pattern Matching (CPM)*, pages 24–35, 2006.

[4] J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[5] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The Connectivity Server: Fast access to linkage information on the Web. In *Proc. 7th World Wide Web Conference (WWW)*, pages 469–477, 1998.

[6] 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.

[7] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. 14th Symposium on Discrete Algorithms (SODA)*, pages 579–588, 2003.

[8] P. Boldi and S. Vigna. The WebGraph framework I: compression techniques. In *Proc. 13th World Wide Web Conference (WWW)*, pages 595–602, 2004.

[9] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web. *Journal of Computer Networks*, 33(1–6):309–320, 2000. Also in *Proc. 9th World Wide Web Conference (WWW)*.

[10] D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsos. Fully automatic cross-associations. In *Proc. ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, 2004.

[11] 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.

[12] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[13] N. Deo and B. Litow. A structural approach to graph compression. In *Proc. of the 23th MFCS Workshop on Communications*, pages 91–101, 1998.

[14] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[15] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[16] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[17] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[18] A. Gulli and A. Signorini. The indexable Web is more than 11.5 billion pages. In *Proc. 14th World Wide Web Conference (WWW)*, pages 902–903, 2005.

[19] X. He, M.-Y. Kao, and H.-I. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *Journal on Discrete Mathematics*, 12(3):317–325, 1999.

[20] X. He, M.-Y. Kao, and H.-I. Lu. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM Journal on Computing*, 30:838–846, 2000.

[21] T. Hu and A. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM Journal of Applied Mathematics*, 21:514–532, 1971.

[22] A. Itai and M. Rodeh. Representation of graphs. *Acta Informatica*, 17:215–219, 1982.

[23] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1989.

[24] K. Keeler and J. Westbook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58:239–252, 1995.

[25] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large scale knowledge bases from the Web. In *Proc. 25th Conference on Very Large Data Bases (VLDB)*, pages 639–650, 1999.

[26] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[27] H.-I. Lu. Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits. In *Proc. 13th Symposium on Discrete Algorithms (SODA)*, pages 223–224, 2002.

[28] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180, pages 37–42, 1996.

[29] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[30] M. Naor. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics*, 28(303–307), 1990.

[31] G. Navarro. Compressing Web graphs like texts. Technical Report TR/DCC-2007-2, Dept. of Computer Science, University of Chile, 2007.

[32] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[33] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proc. 19th International Conference on Data Engineering (ICDE)*, page 405, 2003.

[34] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM 13th Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[35] 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.

[36] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization*, 5(1):47–61, 1999.

[37] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. 11th IEEE Data Compression Conference (DCC)*, pages 213–222, 2001.

[38] G. Turán. Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.

[39] R. Wan. *Browsing and Searching Compressed Documents*. PhD thesis, Dept. of Computer Science and Software Engineering, University of Melbourne, 2003.

[40] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23:337–343, 1977.

[41] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.