

# Extended Compact Web Graph Representations

Francisco Claude<sup>1\*</sup> and Gonzalo Navarro<sup>2\*\*</sup>

<sup>1</sup> David R. Cheriton School of Computer Science, University of Waterloo.

`fclaude@cs.uwaterloo.ca`.

<sup>2</sup> Department of Computer Science, University of Chile. `gnavarro@dcc.uchile.cl`.

**Abstract.** Many relevant Web mining tasks translate into classical algorithms on the Web graph. Compact Web graph representations allow running these tasks on larger graphs within main memory. These representations at least provide fast navigation (to the neighbors of a node), yet more sophisticated operations are desirable for several Web analyses. We present a compact Web graph representation that, in addition, supports reverse navigation (to the nodes pointing to the given one). The standard approach to achieve this is to represent the graph and its transpose, which basically doubles the space requirement. Our structure, instead, represents the adjacency list using a compact sequence representation that allows finding the positions where a given node  $v$  is mentioned, and answers reverse navigation using that primitive. This is combined with a previous proposal based on grammar compression of the adjacency list. The combination yields interesting algorithmic problems. As a result, we achieve the smallest graph representation reported in the literature that supports direct and reverse navigation, and also obtain other variants that occupy relevant niches in the space/time tradeoff.

## 1 Introduction and Related Work

The Web can be modeled as a directed graph: every page corresponds to a node and every link between two pages is represented as a directed edge between the corresponding nodes. This so-called “Web graph” contains an enormous amount of useful information, which is used for a wealth of purposes, from technical (such as improving search engines) to economic (such as detecting potential customers) to scientific (such as carrying out sociological studies).

Methods to discover Web communities, Web spam, Web structure, hubs and authorities, and many others, rely on classical graph algorithms. Donato et al. [16] show how several common Web mining techniques used to discover the structure and evolution of the Web graph build on classical graph algorithms such as depth- and breadth-first-search, reachability, and weakly and strongly connected components. Saito et al. [28] presents a technique for Web spam detection that boils down to algorithms for finding strongly connected components,

---

\* Funded by NSERC of Canada and Go-Bell Scholarships Program.

\*\* Funded in part by Fondecyt Grant 1-080019, and by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

for clique enumeration, and for minimum cuts. A simple representation that allows *direct* navigation (to the nodes pointed from the current one) suffices for these purposes. Yet, there are other important applications where efficient *reverse* navigation (to the nodes pointing to the current one) is also necessary. The HITS algorithm [23] to find hubs and authorities on the Web starts by selecting random pages and finding the induced subgraphs, which are the pages that point to or are pointed from the selected pages [22]. Uniform sampling methods [27] also require direct and reverse navigation, and are usually replaced by suboptimal alternatives due to the difficulty of implementing the latter.

An important limitation when processing this kind of graphs is their size. Many of the graph algorithms we mentioned are not disk-friendly, and thus the sizes of the graphs that can be analyzed by the Web mining applications, and consequently the quality of their results, is limited by the main memory size. Much effort has been spent in representing Web graphs in compressed form, so that the direct neighbors of a node can be efficiently retrieved [10, 1, 30, 8, 13]. Boldi and Vigna [8] achieve currently the least space combined with efficient navigation. In later work [13, 14] we introduced the use of grammar compression of the adjacency lists (more precisely, Re-Pair [24]). This required more space than the best achievable by Boldi and Vigna, but when both methods used the same space, ours was faster. Other techniques [4], instead, achieve even less space than Boldi and Vigna, yet with much higher access times.

By essentially doubling the space of these solutions, one can represent the graph and its transpose, thus providing reverse navigation as well. Needless to say, adding such an amount of redundancy is against the goal of providing a compact representation. The only approach we know of where direct and reverse navigation is supported [9], the  $k^2$ -tree, is based on representing the adjacency matrix in a way that takes advantage of its sparseness. They achieve similar times for direct and reverse queries.

In this work we introduce an alternative way of supporting direct and reverse navigation. Let  $G = (V, E)$  be our graph, where  $n = |V|$  and  $m = |E|$ . We resort to previous work by regarding  $G$  as a *binary relation* on  $V \times V$ , and then use the techniques of Barbay et al. [5], where forward and reverse traversal operations can be solved in time  $O(\log \log n)$  per node delivered. A more recent followup [6] retains those times and reduces the space to the *worst-case* entropy of the binary relation, that is,  $\log \binom{n^2}{m}$  (our logarithms are in base 2).

This worst-case compression, however, is poor for Web graphs, as these are far from random in the Erdős-Rényi sense [17]. To illustrate this, we downloaded four Web crawls from the WebGraph project<sup>1</sup>, which will be used for the experiments along the article. Table 1 shows their main characteristics. The third column shows the size (in bits per edge, bpe) required by a plain adjacency list representation using 4-byte integers. The fourth column shows the space required for a plain representation of the graph plus its transpose. The fifth column shows the lower bound given by the worst-case entropy, and the last column the space actually achieved by the best method based on Re-Pair compression [13]

<sup>1</sup> <http://law.dsi.unimi.it>

Crawl	Nodes	Edges	Plain	2×Plain	Bin.Rel.	Re-Pair
EU	862,664	19,235,140	20.81	41.62	15.25	7.65
Indochina	7,414,866	194,109,311	23.73	47.46	17.95	4.54
UK (2002)	18,520,486	298,113,762	25.89	51.78	20.13	7.50
Arabic	22,744,080	639,999,458	25.51	51.02	19.66	5.53

**Table 1.** Some characteristics of the four crawls used in our experiments, as well as expected space usage (in bpe) with some known methods.

for the direct plus the transposed graph. This shows that the worst-case entropy measure is a poor estimation of the compression that can be achieved.

In this article we combine our previous technique based on Re-Pair [13], which has been successful to compress Web graphs while supporting direct navigation, with the binary relation idea [5]. The latter boils down to representing the adjacency lists using a sequence representation that allows finding the occurrences of a symbol (that is, the places where a given node  $v$  is mentioned in some list), and then find the reverse neighbors using this primitive. The combination with grammar compression poses some interesting algorithmic problems, however, because the compressed text is a sequence of terminals and nonterminals and thus the technique cannot be directly applied. The result achieves forward and reverse navigation within competitive times, and significantly less space, than representing the direct plus the transposed graph using previous techniques. Depending on the compact data structure we use to represent the sequence, we obtain, on one hand, the smallest reported space for a structure that supports bidirectional navigation (indeed, within  $O(\log n)$  time per delivered neighbor); and on the other, a faster ( $O(\log \log n)$  time) and larger data structure that occupies a relevant niche in the space/time tradeoff of the current state of the art.

## 2 Basic Concepts

### 2.1 Compact Data Structures for Sequences

A compact data structure aims at representing the same data as its classical counterpart in little space, while still supporting interesting queries without expanding the whole data structure. Sometimes compact data structures require more time per query in theory, but since they use less space, they can fit in smaller and faster memories. This important advantage allows them to outperform their classical counterparts, especially if we consider the scenario where the classical version of the data structure has to resort to disk, while the compact data structure fits in main memory.

A basic tool used in many compact data structures is the bitmap with *rank* and *select* capabilities [25]. Consider a binary string  $B[1, n]$  We support the following queries:

- $rank_B(b, i)$ : counts how many times the bit  $b$  appears in the prefix  $B[1, i]$ .
- $select_B(b, j)$ : returns the position of the  $j$ -th occurrence of bit  $b$  in  $B$ .
- $access_B(i)$ : retrieves  $B[i]$ .

Clark [12] proposed a solution that achieves constant time for the queries and requires  $n + o(n)$  bits. This was later improved by Raman, Raman and Rao (RRR) [26]. They achieved constant time for the queries while using  $nH_0(B) + o(n)$  bits, where  $H_0$  represents the zero-order entropy. The zero-order (empirical) entropy of a sequence  $S$ , drawn from an alphabet  $\Sigma$  of size  $\sigma$ , is defined as  $H_0(S) = -\sum_{c \in \Sigma} p_c \log p_c \leq \log \sigma$ , where  $p_c = n_c/n$  and  $n_c$  is the number of occurrences of character  $c$  in  $S$ .

*Rank/select/access* queries naturally extend to sequences, where  $b \in \Sigma$ . A sequence representation supporting these primitives is the wavelet tree [20, 25], which achieves  $O(\log \sigma)$  time per query and requires  $n \log \sigma + o(n) \log \sigma$  bits. The wavelet tree stores a number of bitmaps, which can be compressed using RRR, in which case the space requirement drops to  $nH_0(S) + o(n) \log \sigma$  bits.

Golynski, Munro, and Rao (GMR) [18] presented another representation that achieves time  $O(\log \log \sigma)$  for *rank* and *access*, and  $O(1)$  for *select*. Alternatively, they can achieve  $O(1)$  time for *access*,  $O(\log \log \sigma)$  for *select*, and  $O(\log \log \sigma \log \log \log \sigma)$  for *rank*. The structure requires  $n \log \sigma + n o(\log \sigma)$  bits.

Note both structures replace the sequence. Claude and Navarro [15] carried out a practical evaluation of *rank/select/access* capable bitmap and sequence representations. They included a simplified version of GMR for the case  $n \approx \sigma$ , which we call *chunk*. The chunk proved to be very fast for large alphabets, while requiring little extra space on top of  $n \log \sigma$ . Wavelet trees, on the other hand, not only supported the three queries, but also were shown to be an interesting alternative for compressing sequences over large alphabets, where classical methods like Huffman fail due to the alphabet representation overhead. The paper also showed how to omit the pointers of the wavelet tree while preserving its time performance, which saves much space overhead on large alphabets.

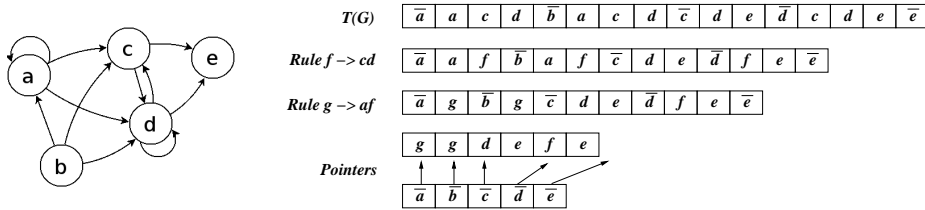
## 2.2 Re-Pair Compression of Web Graphs

Re-Pair [24] is a grammar-based compression algorithm consisting 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. Re-Pair works as follows over a sequence  $L$ : (1) It identifies the most frequent pair  $ab$  in  $L$ . (2) It adds the rule  $s \rightarrow ab$  to a dictionary  $R$ , where  $s$  is a new symbol not appearing in  $L$ . (3) It replaces every occurrence of  $ab$  in  $L$  by  $s$ . (4) It iterates until the replacements do not compensate for the increase of  $R$ .

We call  $C$  the sequence resulting from  $L$  after compression. Every symbol in  $C$  represents a *phrase* (a substring of  $L$ ), which is of length 1 if it is an original symbol (called a *terminal*) or longer if it is an introduced one (a *nonterminal*). Any phrase can be recursively expanded in optimal time (that is, proportional to its length), even if  $C$  is stored on secondary memory (as long as the dictionary of rules  $R$  is kept in RAM).

Re-Pair can be implemented in linear time [24]. However, this requires several data structures to track the pairs that must be replaced. This is problematic when applying it to large sequences. We developed an approximate version [13, 14] that requires little space on top of the sequence.

In our proposal [13] to represent a Web graph  $G$ , each node  $v$  has a special identifier  $\bar{v}$  to mark the beginning of its adjacency list. The representation of the graph,  $T(G)$ , is the concatenation of the representations of all the adjacency lists, defined as  $T(v_i) = \bar{v}_i v_{k_1} v_{k_2} \dots v_{k_r}$  where  $v_{k_j}, 1 \leq j \leq r$ , are the nodes pointed from  $v_i$ . Now  $T(G)$  is compressed using Re-Pair. Since symbols  $\bar{v}_i$  are unique, they stay as terminals in  $C$ . Therefore adjacency lists correspond to substrings in  $C$ , and thus can be decompressed in optimal time. The values  $\bar{v}_i$  are afterwards removed from the sequence, and instead  $n$  pointers to the beginning in  $C$  of the list of each node is stored (in about the same space gained with the removal of the  $\bar{v}_i$ s). This allows direct navigation in optimal time, but not reverse navigation. Later [14], we proposed several variations achieving better space/time. Figure 1 illustrates the process for a small graph.

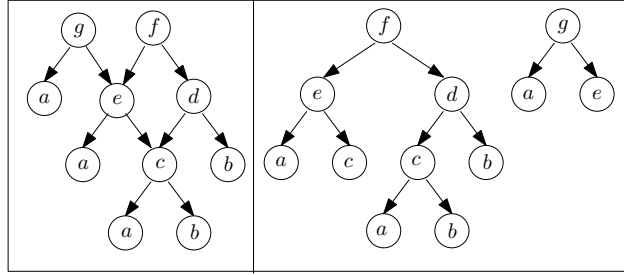


**Fig. 1.** A small example graph, its  $T(G)$  representation, the Re-Pair compression of it, and the final replacement of the  $\bar{v}_i$ s by pointers.

### 2.3 Representing the Re-Pair Rules

The dictionary  $R$  can be represented as an array of pairs of integers, or in some compact form. We use a representation [19] that reduces it to about 50% while retaining efficient access to  $R$ . The set of rules can be seen as a directed acyclic graph of outdegree 2 where internal nodes are nonterminals and leaves are terminals. This is converted into a forest of binary trees, where nonterminal leaves signal shared subtrees. The forest is represented as a sequence  $R_S$  of leaf values and a bitmap  $R_B$  that defines the tree shape. Nonterminals are identified with the starting position of the (sub)tree that defines them in  $R_B$ . In  $R_B$ , the trees are described by a preorder traversal where a 1 represents an internal node (with 2 children) and a 0 represents a leaf. The (terminal or nonterminal) leaf value corresponding to  $R_B[i] = 0$  can be found at  $R_S[\text{rank}_{R_B}(0, i)]$ .

For example, the set of rules  $c \rightarrow ab$ ,  $d \rightarrow cb$ ,  $e \rightarrow ac$ ,  $f \rightarrow ed$  and  $g \rightarrow ae$ , over terminals  $\{a, b\}$ , can be represented as shown in Figure 2. We have  $R_B = 110011000100$  and  $R_S = \mathbf{a6abba2}$ , where the ‘6’ represents the nonterminal ‘c’, whose tree is at position 6 in  $R_B$ ; similarly ‘2’ represents ‘e’.



Position	1	2	3	4	5	6	7	8	9	10	11	12	
$R_B$	=	1	1	0	0	1	1	0	0	0	1	0	0
$R_S$	=			a	6		a	b	b		a	2	

**Fig. 2.** Example of our representation of Re-Pair rules. Top left: the initial DAG. Top right: the forest representation. Bottom: Encoding with  $R_B$  and  $R_S$ .

To expand a given nonterminal at position  $i$  in  $R_B$ , we scan  $R_B[i \dots]$  until we have seen more 0s than 1s, and then collect all the consecutive leaf values. Leaf values corresponding to nonterminals must be recursively expanded.

### 3 A Simple Representation based on Binary Relations

We note that sequence  $T(G)$  (without Re-Pair compression), armed with symbol *rank* and *select* operations, is already able of handling an extended set of queries that includes reverse navigation, using an approach similar to Barbay et al.'s [5]. Assume we store a bitmap  $B$  marking the positions of  $T(G)$  where each adjacency list starts, more precisely, of the positions of the  $\bar{v}_i$ s:  $start(v_i) = select_{T(G)}(\bar{v}_i, 1)$ . We will also use operation  $pred(i) = select_B(1, rank_B(1, i))$  that finds the last 1 up to position  $i$  in  $B$ . We can support the following queries. Note  $|T(G)| = n + m$ .

- outdegree of  $v_i$ : it is  $start(v_{i+1}) - start(v_i) - 1$ .
- the  $k$ -th direct neighbor of  $v_i$ : it is  $T[start(v_i) + k]$ .
- indegree of  $v_i$ : it is  $rank_{T(G)}(v_i, m + n)$ , the number of times  $v_i$  is mentioned in some adjacency list.
- the  $k$ -th reverse neighbor:  $T[pred(select_{T(G)}(v_i, k))]$  gives the corresponding identifier  $\bar{v}$  of the  $k$ -th reverse neighbor.
- edge  $(v_i, v_j)$  exists: if  $rank_{T(G)}(v_j, start(v_{i+1})) - rank_{T(G)}(v_j, start(v_i)) = 1$ .

In practice we remove the  $\bar{v}_i$ s from  $T(G)$  and set  $n$  pointers  $S[v_i]$  to the beginning of each list, so that  $start(v_i) = S[v_i]$  and the  $k$ -th reverse neighbor becomes  $rank_B(select_{T(G)}(v_i, k))$ ,<sup>2</sup> and the slow-in-practice [15]  $select_B$  operation is totally avoided. Array  $S$  requires  $n \log m$  bits of space. Bitmap  $B$  requires

<sup>2</sup> Nodes with zero outdegree must be handled somehow so that they do not interfere with  $rank_B$ , for example by marking them in another bitmap, or renumbering them after all the other nodes.

Crawl	Wavelet Tree	Plain	Bin.Rel.	Re-Pair
EU	13.67	20.81	15.25	7.65
Indochina	14.16	23.73	17.95	4.54
UK	15.05	25.89	20.13	7.50
Arabic	15.30	25.51	19.66	5.53

**Table 2.** Size required by our simple representation, using wavelet trees without pointers and RRR for the bitmaps [15], measured in bpe. The last column adds up the Re-Pair representations for the original and transposed graph.

at most  $mH_0(B) + o(m) = n \log \frac{m}{n} + O(n) + o(m)$  bits using RRR (Section 2.1). The remaining  $T(G)$  can be represented using GMR (Section 2.1), requiring  $m \log n + m o(\log n)$  bits. The total space used is  $m \log n + O(n \log m) + o(m \log n)$  bits. This is basically the space of a plain adjacency list representation, yet we have the extended functionality. On the other hand, the upper bound is higher than the  $\log \binom{n^2}{m} = m \log \frac{n^2}{m} + O(m)$  worst-case entropy of the graph. Operations outdegree, indegree, and reverse neighbors are carried out in constant time per delivered datum<sup>3</sup>, whereas checking existence of edges and retrieving direct neighbors cost  $O(\log \log n)$ .

With a wavelet tree representation, instead, every delivered direct or reverse neighbor (and checking edge existence) takes  $O(\log n)$  time, but  $T(G)$  can be compressed to  $mH_0(T(G)) + o(m) \log n$  bits. Let  $n_i$  be the indegree of node  $v_i$ , thus  $v_i$  appears  $n_i$  times in  $T(G)$ . Then  $mH_0(T(G)) = \sum n_i \log \frac{m}{n_i}$ . Web graphs are known to have varying indegrees: a Zipf-distribution with parameter  $\theta = 2.1$  has been observed [2, 10]. Under this distribution we obtain  $mH_0(T(G)) = c \cdot m \log n + O(m)$ , with  $c = ((\theta - 1) \sum_{i \geq 1} i^{-\theta})^{-1} \approx 0.58$ . On the other hand,  $m/n$  is around 15–30 on Web graphs, so the worst-case entropy is  $\log \binom{n^2}{m} = m \log n - O(m)$ . Table 2 shows that this representation takes less space than a plain adjacency list (which does not answer reverse queries) on our four Web crawls, by a factor remarkably close to 0.58 (except on the smaller EU, where it is 0.65). It also takes less space than the worst-case graph entropy. Still, the last column reminds us that it is still far from the state of the art. In the next section we will combine this idea with Re-Pair compression.

## 4 Combining Re-Pair with Binary Relations

The result of the previous section makes it clear that we cannot go too far with zero-order compression of  $T(G)$  or the graph binary relation. Re-Pair is much more successful. In fact, Re-Pair compression on graphs can be regarded as (and attribute its success to) the decomposition of the graph binary relation into two:

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

<sup>3</sup> For indegree one needs to use other  $n \log m$  bits, otherwise it costs  $O(\log \log n)$ .

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

The regularities exposed by this factorization go well beyond those captured by the worst-case entropy of the original binary relation or zero-order entropy of its sequence representation. In very broad terms, we attempt at representing the graph as the composition of these two binary relations. Using the technique of Barbay et al. [5], each direct neighbor would be retrieved in time  $O(\log \log n)$ , by finding all the Re-Pair symbols that conform its adjacency list (first relation) and then the graph nodes each such symbol expands to (second relation).

Finding the reverse neighbors of node  $v$ , on the other hand, is harder. We should first find all the Re-Pair symbols (nonterminals) that expand to  $v$  (second relation), and then, for each such symbol, all the nodes in which adjacency list the symbol participates (first relation). The problem is that many nonterminals exist in the dictionary for the sake of structuring the grammar but do not appear in  $C$ , and thus we can carry out much work that does not lead to any result.

A further challenge is that representing the second binary relation as such, with the rules in fully expanded form, could require space  $\omega(|R|)$ . Thus we must use the representation of Section 2.3 for the second binary relation, and this complicates the operations we must carry out on it. We describe now our solution.

#### 4.1 Representation

We apply our simple sequence representation of Section 3 on top of the Re-Pair compressed  $T(G)$ , instead of on the plain sequence. We compress  $T(G)$  and represent sequences  $C$  and  $R_S$  not in plain form, but instead using a *rank/select/access* capable representation (see Section 2.1). This can be either:

- The GMR representation. It does not compress  $C$  or  $R_S$  any further, but it provides *access* and symbol *rank* in time  $O(\log \log n)$  (yet typically constant), and symbol *select* in constant time.
- A wavelet tree. The operations are carried out in  $O(\log n)$  time, but the representation compresses further  $T(G)$  up to its zero-order entropy. Albeit significantly slower, this achieves unprecedented space results, as we see later.

Extraction of the direct neighbors is done exactly as in previous work [13], using *access* on the sequences  $C$  and  $R_S$  to expand the list of the desired node.

#### 4.2 Extracting Reverse Neighbors

As explained, to find reverse neighbors of  $v$  we must consider that  $v$  may appear not only explicitly in  $C$ , but also implicitly, in the form of a nonterminal that expands to  $v$ . Given our representation of the rules  $R$ , we must look for  $v$  in  $R_S$  and, for each occurrence, collect all of its ancestors in  $R_B$ , and look for each of them in  $C$ . Because each such ancestor might appear in  $R_S$  again (due to the conversion of the DAG into a forest) and have further ancestors, the process has to be repeated recursively for every ancestor found.

---

```

rev-adj( $v$ )
1.  For  $k \leftarrow 1$  to  $rank_C(v, |C|)$  Do
2.       $occ \leftarrow select_C(v, k)$ 
3.      report  $rank_B(1, occ)$ 
4.  For  $k \leftarrow 1$  to  $rank_{R_S}(v, |R_S|)$  Do
5.       $occ \leftarrow select_{R_S}(v, k)$ 
6.      For each  $s$  ancestor of  $R_S[occ]$  in  $R_S$  Do
7.          rev-adj( $s$ )

```

---

**Fig. 3.** Obtaining the reverse adjacency list

The occurrences of  $v_i$  (or, recursively, any other nonterminal) in  $R_S$  are obtained using  $select_{R_S}(v_i, k)$ . In order to extract the ancestors, we can use an alternative representation for  $R_B$  called LOUDS [21]. LOUDS represents each leaf with a 0 and has been shown to be very effective when only parent/child traversals are required [3], which makes it ideal for our purpose. We adapt LOUDS to binary forests as follows. Let  $f$  be the number of trees in the dictionary forest. The forest is traversed level-wise and left-to-right within each level, and for each node found we write a 1 if the node has (two) children and a 0 if not. In Figure 2,  $f = 2$  and  $R_B = 111100001000$ . Each node is identified with its corresponding bit position  $i \geq 1$ . Now, LOUDS formulas become as follows:

- $child_{left/right}(i) = f - 1 + 2 \cdot rank_{R_B}(i, 1) + 0/1$ ,
- $parent(i) = select_{R_B}(\lfloor (i - f + 1)/2 \rfloor)$  ( $i$  is a root if  $i \leq f$ ).

We call this solution GMR LOUDS\*. Although constant-time in theory, this solution resorts to *select* on bitmaps, which is not that fast in practice [15]. An alternative, less sophisticated, solution is to mark the beginning of the top-level trees of  $R_B$  in another bitmap. Then we unroll the whole tree containing the occurrence in  $R_S$  and spot the ancestors. We call this second solution GMR.

Figure 3 shows the algorithm for retrieving the reverse neighbors. We use the bitmap  $B$  that marks the beginning of each adjacency list in  $C$ . This bitmap is included in the original structure [14], so it does not add any more space and allows us to determine to which list a position in  $C$  belongs.

No reverse neighbor is reported twice: Even if we find several times the same position of  $C$  along the process, it will be for different occurrences within  $C$ .

On the other hand, we are not providing any time guarantee for the process, because as explained we might do sterile work for ancestors in  $R_B$  which do not appear in  $C$ . For the others we obtain at least one occurrence per access to the sequences. We address this problem next.

## 5 Guaranteeing Reverse Neighbor Retrieval Time

A way to alleviate the problem in practice is to include a bitmap, parallel to  $R_B$ , which indicates, for each internal node, whether or not it appears in  $C$  or in  $R_S$ . This can be combined either with the LOUDS or the basic representation of

$R_B$ . Each time we find an ancestor, the parallel bitmap indicates immediately whether it is worth paying the effort of looking for it in  $C$  and  $R_S$ . Indeed, given the negligible cost of such a bitmap, we opt for storing two of them: one referring to  $C$  and the other to  $R_S$ . This helps us limiting further unnecessary searches, although we still pay a constant-time cost to process useless nodes. We will test these bitmaps in combination with binary-tree LOUDS (GMR LOUDS\* M), with general LOUDS (GMR LOUDS M, just to test it loses to the previous one), and with the basic representation (GMR M).

An alternative to achieve a logarithmic-time guarantee per retrieved neighbor is to use balanced Re-Pair [29], which enforces logarithmic rule heights. Since the roots of the DAG must appear in  $C$ , in the worst case we pay  $O(1)$  time to discard (using the bitmaps introduced above) each element of an upward path (of length now limited to  $O(\log m)$ ), except the root. As the root yields at least one neighbor, we can charge this  $O(\log m)$  time to that result.

A solution that guarantees a constant number of operations on the sequences per reverse neighbor delivered is to effectively remove from the forest those nodes that do not appear in  $C$  nor in  $R_S$ . The children of the removed node become children of their former grandparent. This can be done precisely because those nodes will not be accessed from elsewhere. The result is not anymore binary, but a general tree that is represented with the original LOUDS format [21, 3]. Now we can prove our result.

**Lemma 1.** *Using the reduced tree, we pay  $O(1)$  operations on the sequences per reverse neighbor delivered.*

*Proof.* Consider the original DAG with the useless nodes removed. Then each node either (a) is a root, (b) appears in  $C$ , or (c) has at least two parents (i.e., appears again in  $R_S$ ). The algorithm in Figure 3 is equivalent to starting from some arbitrary node and traversing the DAG upwards, so that a constant number of operations on  $R_S$ ,  $R_B$  and  $C$  are carried out (i) per DAG node considered and (ii) per result retrieved. We focus on (i). Nodes of type (a) and (b) yield at least one result, so their cost can be absorbed by (ii). For nodes of type (c), they have at least two parents, and thus each unit of work invested on them increases at least by 1 the number of results to report.  $\square$

Therefore, combined with GMR representation, we recover the constant time per reverse neighbor we had in Section 3. This representation is called GMR LOUDS in the experiments.

## 6 Experimental Results

The experiments were run on a 2GHz Intel Xeon (8 cores) with 16 GB RAM, running Ubuntu GNU/Linux with kernel 2.6.22-14 SMP (64 bits). The code was compiled with g++ using the -O9 directive.

From the several Re-Pair based versions studied in previous work [14], we chose “Reord CDict NoPtrs”. For the wavelet trees we used the version without

Crawl	Re-Pair WT	Re-Pair GMR	Re-Pair (dir+rev)	$k^2$ -tree	WebGraph (dir+rev)	Asano $\times 2$
EU	3.93	5.86	7.65	5.20	7.20	5.56
Indochina	2.30	3.65	4.54	2.82	2.94	
UK	3.98	6.22	7.50	4.20	4.34	
Arabic	2.72	4.15	5.53		3.25	

**Table 3.** Space consumption (in bpe) of the Re-Pair based compressed representations of the adjacency lists, and previous work.

pointers [15]. For the GMR structure we used the simpler and faster *chunk* variant [15], as the grown alphabet after running Re-Pair on  $T(G)$  (originally  $n$ ) and the reduced length (originally  $m$ ) become sufficiently similar.

We first focus on achieving minimum space usage, while still retrieving direct and reverse neighbors within reasonable time, that is, much faster than decompressing the whole graph. Later we focus on the faster alternatives. The space we report does not include special bitmaps for computing in/outdegrees.

Table 3 shows the space required for the four crawls. The first two columns are our contributions. In column **Re-Pair WT** we represent  $C$  and  $R_S$  using a compressed wavelet tree with sample value 64 [15] (space decreases by about 0.20 bpe more with larger sample values, but retrieval times degrade). In column **Re-Pair GMR** we show the representation that combines Re-Pair with a GMR *chunk*. Next columns are previous alternatives. Column **Re-Pair** shows the space needed by Re-Pair compression (variant “Diffs CDict NoPtrs”) of the graph plus its transpose (so as to support direct and reverse queries) [13]. Column  $k^2$ -tree gives the smallest space achieved by that technique [9] (the space for the largest graph, **Arabic**, is not reported in there, and we could not build it either).

Column **WebGraph** gives the space achieved by the *WebGraph* technique [7], version 2.4.2, using variant **strictHostByHostGray**, which gave the best results. We add up the space for the direct and the transposed graph. We account only the space the structure requires on disk, even if the process requires much more memory to run. On the other hand, we account for their “offset” structure, which is the one providing direct access to the neighbors (without the offsets, the structure degenerates into a pure compression scheme). For this experiment we set the parameters so as to largely favor compression over speed (window size 10, maximum reference unlimited). With this compression they retrieve direct neighbors in about 100 microseconds.

Finally, column **Asano $\times 2$**  shows the space achieved by Asano et al. [4] on the EU graph (which is the largest graph they report). We double the space to account for the transposed graph. The time they report is over 1 millisecond per neighbor retrieved, whereas typical times (as shown next) are a few microseconds. Doubling the space is a bit pessimistic, as the transposed graph compresses slightly better, but still the difference with **Re-Pair WT** is significant, and this was the only reasonable way we found to try including them in the comparison.

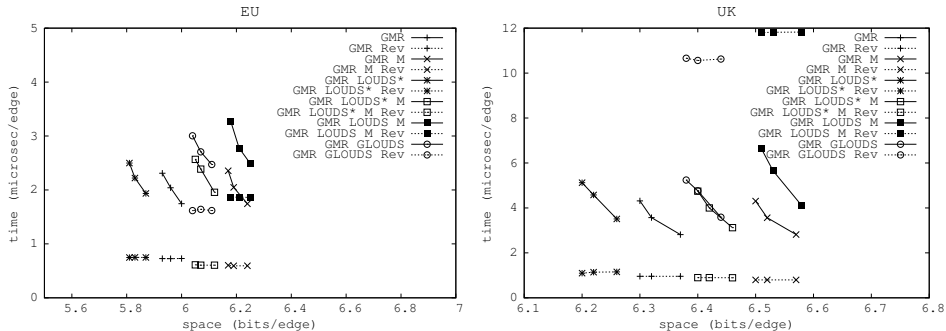


Fig. 4. Space/time tradeoffs of the different dictionary representations.

We observe that our techniques require less space than adding up direct and reverse Re-Pair compressed graphs, while achieving good performance, as we see soon. By combining with a wavelet tree, on the other hand, we achieve *the smallest space reported in the literature* while supporting direct and reverse neighbors in reasonable time: around 35 microseconds/edge for direct and 55 for reverse neighbors. The next experiments show that, using more space, one can reduce these times by an order of magnitude. However, no alternative scheme can operate within tens of microseconds and achieve the space of Re-Pair WT.

Figure 4 shows direct and reverse neighbor retrieval times on two crawls, for the different alternatives studied in Sections 4.2 and 5. Reverse retrieval times are marked **Rev**. As it can be seen, the general LOUDS versions (without modifier “\*”) lose to the simpler ones, and also the idea of marking nodes (suffix “M”) does not pay off. The space/time map is dominated by GMR and GMR LOUDS\*. We use GMR for the rest of the experiments.

Figure 5 shows retrieval times obtained for the four crawls, for both forward and reverse neighbors. We include only the techniques that are most competitive in time: **WebGraph** (storing both direct and reverse graphs), **Re-Pair** (storing both direct and reverse graphs), **Re-Pair GMR** (ours), and  $k^2$ -trees (variants called **Hybrid5** and **Hybrid37**, which give the best space/time tradeoffs [9]). We also include a variant of **Re-Pair GMR** labeled “(2)”, where we use the variant of GMR that solves *access* in  $O(1)$  time and *select* in time  $O(\log \log n)$ . Thus, while **Re-Pair GMR** is faster for reverse neighbors (using constant-time *select*), **Re-Pair GMR (2)** is faster on direct neighbors (using constant-time *access*)<sup>4</sup>. When times are not constant, an internal sampling used to compute an inverse permutation produces the observed space/time tradeoff.

**Re-Pair GMR** is not as fast as **Re-Pair** (at best, 3 times slower), but it requires significantly less space (about 25%). The  $k^2$ -tree (with variant **Hybrid5**) can

<sup>4</sup> Alternatively, we could have used the original structure and index the transposed graph, but this turned out not to be a good idea: compression of the reverse graph generates many more dictionary symbols and deeper dictionary trees, and thus both queries are slower than on **Re-Pair GMR (2)**.

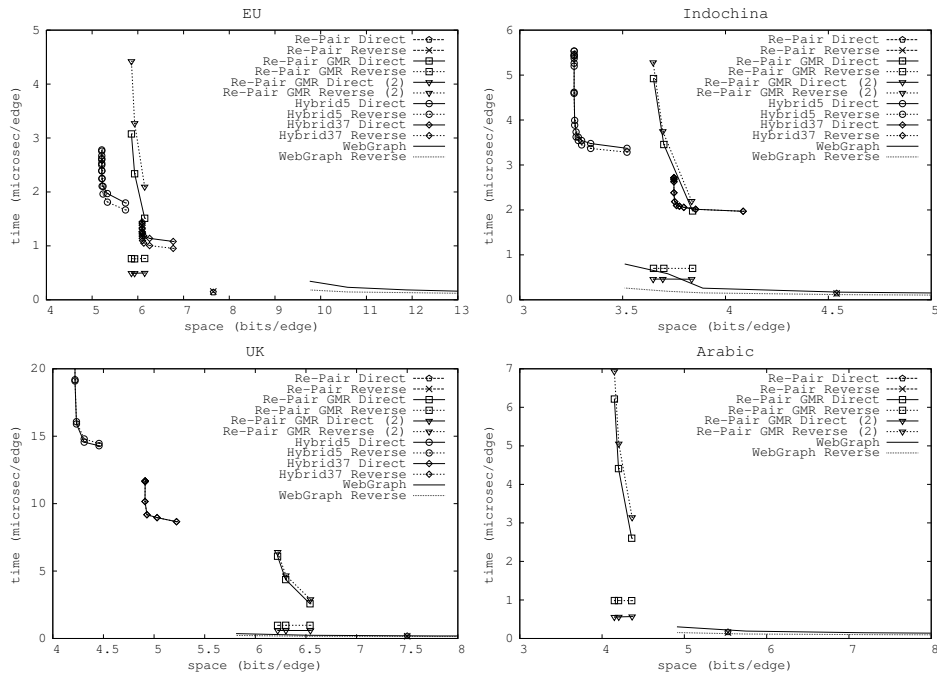


Fig. 5. Space/time tradeoffs of the most competitive variants.

achieve about 13% less space than the second point of **Re-Pair GMR** (recall that  $k^2$ -tree used much more space than **Re-Pair WT**, but it is much faster than it). Yet, when using that space, it is either 4–7 times slower for direct neighbors and 1.0–1.5 times faster for reverse neighbors (if using our variant (2)), or about 3–5 times slower for reverse neighbors and similar for direct neighbors (if using our first variant). A nice point in  $k^2$ -tree is that it is symmetric (in technique and time) to obtain forward or reverse neighbors. A nice point in our structure is that, if one is interested mainly in direct or reverse neighbors, one can choose one of the two alternatives and be much faster on those queries, while still supporting the others in reasonable time. Adding up both times, we see that our alternative would be very close to **Hybrid37**, in space and time, if both direct and reverse neighbors had to be obtained. (An exception is graph **UK**, where  $k^2$ -tree is 35% smaller than **Re-Pair GMR**, but significantly slower in all aspects.)

For **WebGraph** we show the curves reaching as much as possible to the left; using less space yields a sudden increase in time. The comparison with our technique is mixed. On **EU** and **Arabic**, **WebGraph** cannot approach the space we use (while maintaining reasonable retrieval performance). On **Indochina**, both achieve comparable results. On **UK**, instead, **WebGraph** dominates.

## 7 Conclusions

We introduced a technique to represent Web graphs in compressed form so that not only fast access to the (direct) neighbors is supported, but also to the reverse neighbors (that is, nodes pointing to a given one). This has many applications to several Web analysis and mining tasks, where the memory limitations pose serious obstacles to analyzing massive graphs. Our representation combines grammar-based compression with compact data structures for sequences that represent the compressed adjacency lists relations, and for trees that represent the grammar DAG. We provide several solutions, which support direct and reverse neighbor retrieval within a time that ranges from constant to logarithmic.

We achieve several relevant space/time tradeoffs. On one hand, we achieve the most compact functional Web graph representation reported up to date. On a sample of Web crawls, it required 2.3–4.0 bits per edge (bpe) while supporting direct and reverse navigation within a few tens of microseconds per neighbor. The best alternatives require 2.8–5.2 bpe for the same functionality. Compared to a  $2 \times 32$ -bit plain representation of the graph plus its transpose, we allow handling graphs 15–30 times larger within the same main memory.

If slightly more space is available, our faster representation requiring 3.6–6.5 bpe is of interest. It supports direct and reverse navigation within 1–3 microseconds per neighbor, occupying a relevant niche among alternative representations.

It would be of interest to extend this research to the compression of other types of networks with similar characteristics. For example, compression of social networks is starting to receive attention [11]. These share some characteristics with Web graphs, yet they have other unique ones such as reciprocity in links and presence of relatively large cliques or bicliques. In particular, many social networks are undirected. With current techniques, the representation of an undirected graph forces either to duplicate each edge  $\{u, v\}$  as  $(u, v)$  and  $(v, u)$ , or to choose arbitrarily from both, but then the (undirected) neighbors of  $v$  will be the union of its direct and reverse neighbors under this representation. Data structures like ours are ideal for this scenario.

## References

1. Adler, M., Mitzenmacher, M.: Towards compressing Web graphs. In: Proc. 11th DCC. pp. 203–212 (2001)
2. Aiello, W., Chung, F., Lu, L.: A random graph model for massive graphs. In: Proc. 32th STOC. pp. 171–180 (2000)
3. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. 11th ALENEX (2010), to appear
4. Asano, Y., Miyawaki, Y., Nishizeki, T.: Efficient compression of Web graphs. In: Proc. 14th COCOON. pp. 1–11. LNCS 5092 (2008)
5. Barbay, J., Golynski, A., Munro, I., Rao, S.S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. In: Proc. 17th CPM. pp. 24–35 (2006)
6. Barbay, J., He, M., Munro, I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proc. 18th SODA. pp. 680–689 (2007)

7. Boldi, P., Santini, M., Vigna, S.: Permuting web graphs. In: Proc. 6th WAW. pp. 116–126. LNCS 5427 (2009)
8. Boldi, P., Vigna, S.: The WebGraph framework I: compression techniques. In: Proc. 13th WWW. pp. 595–602 (2004)
9. Brisaboa, N., Ladra, S., Navarro, G.:  $k^2$ -trees for compact web graph representation. In: Proc. 16th SPIRE. pp. 18–30. LNCS 5721 (2009)
10. Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.: Graph structure in the Web. *Journal of Computer Networks* 33(1–6), 309–320 (2000)
11. Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., Raghavan, P.: On compressing social networks. In: Proc. 15th KDD. pp. 219–228 (2009)
12. Clark, D.: Compact Pat Trees. Ph.D. thesis, University of Waterloo (1996)
13. Claude, F., Navarro, G.: A fast and compact Web graph representation. In: Proc. 14th SPIRE. pp. 105–116. LNCS 4726 (2007)
14. Claude, F., Navarro, G.: Fast and compact Web graph representations. Tech. Rep. TR/DCC-2008-3, Dept. of Comp. Sci., Univ. of Chile (2008)
15. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Proc. 15th SPIRE. pp. 176–187. LNCS 5280 (2008)
16. Donato, D., Laura, L., Leonardi, S., Meyer, U., Millozzi, S., Sibeyn, J.: Algorithms and experiments for the Web graph. *Journal of Graph Algorithms and Applications* 10(2), 219–236 (2006)
17. Erdős, P., Rényi, A.: On random graphs I. *Publications Mathematicae* 6, 290–297 (1959)
18. Golynski, A., Munro, I., Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th SODA. pp. 368–373 (2006)
19. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Proc. 18th CPM. pp. 216–227. LNCS 4580 (2007)
20. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA. pp. 841–850 (2003)
21. Jacobson, G.: Succinct Static Data Structures. Ph.D. thesis, Carnegie Mellon University (1989)
22. Kleinberg, J., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: The Web as a graph: Measurements, models, and methods. In: Proc. 5th COCOON. pp. 1–17. LNCS 1627 (1999)
23. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. *Journal of the ACM* 46(5), 604–632 (1999)
24. Larsson, J., Moffat, A.: Off-line dictionary-based compression. *Proc. IEEE* 88(11), 1722–1732 (2000)
25. Navarro, G., Mäkinen, V.: Compressed full-text indexes. *ACM Computing Surveys* 39(1), article 2 (2007)
26. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding  $k$ -ary trees and multisets. In: Proc. 13th SODA. pp. 233–242 (2002)
27. Rusmevichientong, P., Pennock, D., Lawrence, S., Giles, C.L.: Methods for sampling pages uniformly from the World Wide Web. In: Proc. AAAI Fall Symposium on Using Uncertainty Within Computation. pp. 121–128 (2001)
28. Saito, H., Toyoda, M., Kitsuregawa, M., Aihara, K.: A large-scale study of link spam detection by graph algorithms. In: Proc. 3rd AIRWeb (2007)
29. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms* 3(2-4), 416–430 (2005)
30. Suel, T., Yuan, J.: Compressing the graph structure of the Web. In: Proc. 11th DCC. pp. 213–222 (2001)