Fast and Compact Web Graph Representations

Francisco Claude
fclaude@cs.uwaterloo.ca
David R. Cheriton School of Computer Science
University of Waterloo

Gonzalo Navarro
gnavarro@dcc.uchile.cl
Department of Computer Science
University of Chile

May 7, 2010

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. The state of the art is well represented by the *WebGraph* project, where advantage is taken of several particular properties of Web graphs to offer a tradeoff between space and access time. In this paper we show that the same properties can be exploited with a different and elegant technique that builds on grammar-based compression. In particular, we focus on Re-Pair and on Ziv-Lempel compression which, although cannot reach the best compression ratios of *WebGraph*, achieve much faster navigation of the graph when both are tuned to use the same space. Moreover, the technique adapts well to run on secondary memory and in distributed scenarios. As a byproduct, we introduce an approximate Re-Pair version that works efficiently with severely limited main memory.

1 Introduction

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 graph is used to gather information about the Web, for example, to characterize its shape, prioritize crawling, discover communities, etc.

Many techniques of interest to obtain information from the Web structure are essentially basic algorithms applied over the Web graph. One of the classical references on this topic [KKR+99] shows how the HITS algorithm to find hubs and authorities on the Web [Kle99] starts by selecting random pages and finding the induced subgraphs, which are the pages that point to or are pointed from the selected pages. [DLL+06] show that several common Web mining techniques used to discover the structure and evolution of the Web graph build on classical graph algorithms such as depth-first search (DFS), breadth-first-search (BFS), reachability, and weakly and strongly connected components. [STKA07] present a technique for Web spam detection that boils down to algorithms for finding strongly connected components, for clique enumeration, and for minimum cuts. There are entire conferences devoted to graph algorithms for the Web (e.g. *WAW: Workshop on Algorithms and Models for the Web-Graph*).

In order to efficiently support these algorithms and traversals, one needs to provide a data structure that retrieves the neighbors of a given node, or in the case of Web crawls, the pages pointed from a given page. An important limitation when processing this kind of graphs is their size. For example, according to *WorldWideWebSize*[1], the graph of the Web indexed by *Yahoo!*, *Google*, *Bing*, and *Ask*, is estimated to have at least 55 billion pages. Considering the typical number of outlinks per page, this amounts to at least 1 trillion edges. A plain adjacency list representation of this graph would need around 4 TB of memory space. Three kinds of approaches have been tried to manage huge graph traversals:

- Represent the graph in external memory [Vit06]: Using suitable memory layouts, several graph traversal algorithms run I/O-optimally on disk. Under the *semi-external model* (where the array of nodes stays in main memory and the edges on disk), BFS/DFS take $O(m/B)$ I/O operations when $n < M$, being n the number of nodes, m the number of edges, B the size of a disk page, and M the size of the main memory (B and M measured in number of memory words).

  Since a disk access can be up to 10^6 times slower than a main memory access, these algorithms will certainly perform much worse than the version in main memory, even if I/O-optimal. Yet their advantage is that they can manage huge graphs at low cost, since external memory is much cheaper than main memory.

- Using distributed systems [BBYRNZ01, TGM93]: Distributing the information among many computers is a good solution to manage huge amounts of data, in the aggregated main memory of all the machines. Still, depending on the problem, the communication between the machines may pose a significant latency, comparable to disk times in some cases.

- Compressed data structures [NM07]: The aim is to represent the data in compressed form while retaining the ability to answer the same queries as their uncompressed counterpart. Even if those structures are several times slower than their uncompressed version, they are still orders of magnitude faster than operating the data on secondary memory.

In this paper we focus on the latter approach. We aim at representing graphs in highly compressed form, so as to manage huge instances in main memory. We show that, for example, a 5-billion-edge crawl can be efficiently handled within a main memory of 2 GB (whereas a plain representation would require 22 GB). For larger graphs, where compression is not sufficient to fit them in RAM, compressed data representations have the potential of improving the other two approaches as well. For secondary memory data structures, if one reduces the space required by the data on disk, and keeps locality of access, the net effect is a reduction of m in the $O(m/B)$ time formula, due to reduced seek and transfer time. For distributed computing, compressed data structures may allow using fewer computers to do the same task, and reducing network traffic as well. Therefore, research in compressed data structures to handle Web graphs is useful regardless of the approach.

As far as we know, the best space/time tradeoffs to compress Web graphs such that they can be navigated in compressed form are those of Boldi, Santini, and Vigna [BV04a, BSV09]. 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, so as to offer an excellent tradeoff between compression ratio and time to access the list of neighbors of a node. For this sake they resort to several mechanisms such as node reordering, differential encoding, compact interval representations and references to similar adjacency lists. They developed and maintain a so-called

[1]`http://www.worldwidewebsize.com`

WebGraph framework. It is associated to the site `http://webgraph.dsi.unimi.it`, which by itself witnesses the level of maturity and sophistication that this research area has reached.

In this paper we present a new way of taking advantage of the regularities that arise in Web graphs. Instead of different ad-hoc techniques, we use a uniform and elegant technique called Re-Pair [LM00] to compress the adjacency lists. Re-Pair recursilvey finds pairs of repeated symbols across all the lists and condenses them into a new "nonterminal" symbol, which has to be expanded later when extracting the list. As the original linear-time Re-Pair compression requires much main memory (2 to 5 integers per edge), 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 for compressing huge sequences of any kind.

Our experimental results over different Web crawls show that, although our methods cannot reach the best compression ratios currently achieved within *WebGraph*, our traversal is 1.5–2 times faster when we leave the *WebGraph* representations use as much memory as we need. 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. For larger graphs, as explained, our technique allows reducing seek times. For example, based on our compression results, we extrapolate that the trillion-edge whole-indexed-Web estimation could be accessed up to 10 times faster on disk, requiring just 20 to 45 GB (in RAM) for the nodes and 350 GB (on disk) for the edges, in the semi-external memory model. This amount of RAM is becoming feasible on commodity servers.

From a more general perspective, we advocate for using grammar-based compression techniques to compress Web graphs. These compressors find repeated subsequences and replace them by new (so-called nonterminal) symbols. 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 [ZL78] 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. Both methods can be seen as approximations to the smallest grammar generating the graph. Finding such smallest grammar is NP-hard [Ryt03, CLL$^+$05]. Existing approximations [Ryt03, Sak05] require much space at compression time, which makes them infeasible for our application.

A conference version of this paper appeared in 2007 [CN07]. Since then, other approaches have been proposed that can be regarded as advocating for grammar-based compression. [BC08] introduced the idea of "mining virtual nodes". Translated into our terminology, their idea is to find groups of (not necessarily consecutive) nodes that appear in several adjacency lists, replacing them by a new symbol representing a virtual node, and iterating. By identifying virtual nodes with nonterminals in our grammars, we have that their approach can be seen also as grammar-based compression (especially because order is not important within adjacency lists, and thus putting together the symbols to replace is valid). Their techniques to mine virtual nodes can be regarded, in this framework, as yet another heuristic trying to solve the smallest grammar problem. We show that simple Re-Pair is competitive with this promising line of research.

The source code for a representative subset of the variants proposed in this paper can be downloaded from `http://webgraphs.recoded.cl/`, we also include some examples and further documentation.

2 Related Work

The related work is divided into two parts. The first covers graph representations, starting with a short survey for general graphs and then focusing on Web graphs. The second part is related to compressed data structures. We focus mainly on *rank* and *select* queries over sequences, introducing at the same time the notion of entropy for sequences.

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 $n \log(2m) + 2m \log n$ bits[2], respectively, for undirected graphs. For directed graphs the numbers are n^2 and $n \log m + m \log n$, respectively. We call the *neighbors* of a node $v \in V$, those $u \in V$ such that $(v, u) \in E$, which in our application correspond to the Web pages pointed by v.

The first compressed data structure for graphs we know of [Jac89] requires $O(gn)$ bits of space for a g-page graph (here a "page" is a subgraph whose nodes can be written in a linear layout so that its edges do not cross) . The t neighbors of a node can be retrieved in $O(g + t \log n)$ time. The main idea is 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 time was later improved to $O(g + t)$ by using improved parentheses representations [MR97], and also the constant term of the space complexity was improved [CGH$^+$98]. 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.

Those techniques based on number of pages, as well as many others for planar and geometric graphs we are omitting, are unlikely to perform well on 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 [DL98, HKL00, CPMF04] (yet not supporting access to the compressed graph), the best results are achieved in recent work [BBK03, Bla06]. 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.

[Bla06] applies the separator technique to design a compressed data structure that gives constant access time per delivered neighbor. The technique is carefully implemented and experimented on several graphs. In particular, on a graph of 1 million (1M) nodes and 5M edges from the Google programming contest[3], the data structures require 13–16 bits per edge (*bpe*; this is the total bits divided by the number of edges), 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 [BKM$^+$00, AM01, SY01, BV04a, BSV09]. 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,

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

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

that is, the fraction of pages having i links is $1/i^\theta$ for some parameter $\theta > 0$. Different experiments give rather consistent values of $\theta = 2.1$ for incoming and $\theta = 2.72$ for outgoing links [ACL00, BKM+00].

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 [BBH+98]. 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 [KRRT99, BV04a]. This permits compressing them by a reference to the similar list plus a list of edits.

[SY01] 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 [RGM03]. Different authors [AM01, RSWW01] 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. [BV04a] built on previous work [AM01, RSWW01] and further engineered the compression to exploit the properties above. They have continued improving their scheme within the *WebGraph* framework, and currently display the best tradeoffs between space usage and access time [BSV08, BSV09].

Experimental figures are not 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, [SY01] require 17.83 bpe. [RSWW01], over a graph of 61M nodes and 1G edges, achieve 5.07 bpe for the graph. [AM01] achieve 8.3 bpe over TREC-8 Web track graphs (WT2g set), yet they cannot access the graph in compressed form. [BKM+00] require 37.87 bpe on a graph of 200M nodes and 1.5G edges (and can answer reverse neighbor queries as well). [BSV08]BSV09 largely improve upon those results. For example they report in their *WebGraph* site, on a 133M node and 5.5G link crawl, compression to slightly more than 2.6 bpe. In our experiments on this paper we show they achieve reasonable traversal times within this space, if we use their better variant [BSV09]. In all the representations that offer efficient access times [SY01, RGM03, RSWW01, BSV09], these are of a few hundred nanoseconds per delivered edge.

[BC08] achieve the best compression we are aware of for massive graphs, by exploiting structural properties of Web graphs and social networks. Specifically, they look for bi-cliques, that is, pairs A and B of sets of nodes such that each node in A points to all nodes in B. Then they create a "virtual" node so that all nodes in A point to it and it points to all the nodes in B. This is applied iteratively until no good bi-cliques are found. Several heuristics are tried to find the bi-cliques, which is a hard problem. They do not give times to extract neighbors, yet these are probably competitive (albeit slower than those we achieve in this paper) as they have to decode the integers (they use ζ-codes [BV04b]) and then expand nonterminals. We note that [AMN08] report even better compression figures by exploiting frequent patterns in the adjacency matrix, but their method does not support efficient extraction of edges and is difficult to apply on large graphs (the largest figure reported is for a 20M edge graph, where they achieve 2.78 bpe, whereas [BC08] achieve 2.90 bpe).

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) [Cla96, Mun96]. An improvement to this solution [RRR02] 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.

[GMR06] presented a data structure capable of performing these three operations in a sequence $S[1, n]$ using $n \log \sigma + n\, o(\log \sigma)$ bits and $O(\log \log \sigma)$ time. Note that $n \log \sigma$ is the space required by a plain representation of the sequence. [FMMN07] 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* [GGV03]. 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 [RRR02], the wavelet tree over the sequence $S[1, n]$ requires overall space $nH_0(S) + o(n) \log \sigma$ bits. It implements *access*, *rank*, and *select* via $\log \sigma$ constant-time *rank/select* operations on the bitmaps. [FMMN07] improve upon this result by using multiary wavelet trees.

3 Re-Pair and Our Approximate Version

Re-Pair [LM00] 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 (s is called a *nonterminal*).
3. It replaces every occurrence of ab in T by s.[4]

[4]In case of overlaps one replaces greedily left-to-right, e.g., one cannot replace both occurrences of aa in aaa, so one replaces the first pair.

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 (called *terminals*).

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 [LM00] 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 [GN07], 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 [LM00]. However, this requires several data structures to track the pairs that must be replaced. These require too much space and non-local accesses, so compressing large sequences is problematic. This has been noted in applications of Re-Pair to natural language text compression [Wan03], and suffix array compression [GN07], where workarounds specific of those applications were devised. In the first case, the sequence was compressed by chunks and a complex postprocessing for merging dictionaries was applied. In the second, an efficient approximate version that used specific properties of suffix arrays was introduced, yet it cannot be applied in general.

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 accuracy 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 $M > |T|$ units of main memory available, that is, the text fits in main memory. Section 3.3 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, using a heap of k pointers to cells in H. 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). Special care is needed to handle this problem.

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, but wait to be 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 symbol. 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.

3.2 Analysis.

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 consider $|R|$ 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 accuracy by tuning k. The larger k, the faster the algorithm (as there is an $O(\log(k)/k)$ factor in its time complexity), but the less similar the result compared to the exact method. Note that the algorithm considers carrying out $|R|$ replacements, but some can be disregarded after taking into account the impact of other simultaneous replacements. Thus the final number of rules can be less than the number $|R|$ initiall considered.

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.

3.3 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* ($\not\in \{null, proceed\}$) 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 $O(p \cdot (n/B + k))$, 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 $k \leq n/B$, which for large graphs is a reasonable limit.

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, and where we impose that all the elements listed inside an adjacency list must be sorted by their id. Figure 1 shows an example graph, and Figure 2 illustrates the construction of the structure using the original Re-Pair algorithm.

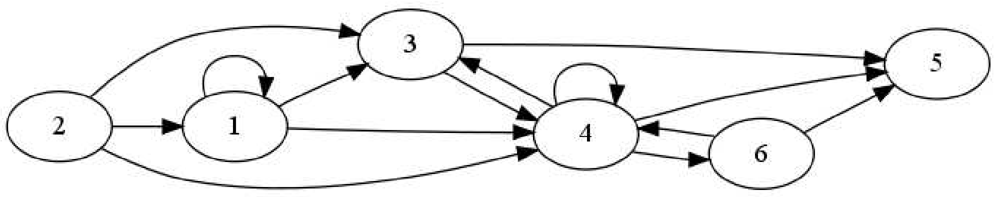

Figure 1: An example graph.

T(G) =	-1	1	3	4	-2	1	3	4	-3	4	5	-4	3	4	5	6	-5	-6	4	5
Add rule 7 → 4 5	-1	1	3	4	-2	1	3	4	-3	7	-4	3	7	6	-5	-6	7			
Add rule 8 → 1 3	-1	8	4	-2	8	4	-3	7	-4	3	7	6	-5	-6	7					
Add rule 9 → 8 4	-1	9	-2	9	-3	7	-4	3	8	6	-5	-6	7							
Remove < 0	9	9	7	3	8	6	7													
	1	*2*	*3*	*4*	*5*	*6*	*7*													

Ptrs[1] = 1
Ptrs[2] = 2
Ptrs[3] = 3
Ptrs[4] = 4
Ptrs[5] = 7
Ptrs[6] = 7

Removing Pointers

B_1 = 111101
B_2 = 1111001

Figure 2: The result of compressing the text representing the graph of Figure 1. We show the resulting text after three replacements and then we remove the delimiters. Just below the resulting sequence we show the pointers to each adjacency list and the corresponding bitmaps obtained when we remove to those pointers to improve the space. This last idea is explained in Section 4.1. Values $\bar{v}$ are represented as $-v$.

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.
- 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 (the set of rules) 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 [GN07] (Section 3) 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 [BV04a] 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 [BV04a] achieve by explicitly encoding the differences using gaps, bitmaps, and other tools.
- The locality property (i.e., the fact that most outgoing links from each page point within the same domain) is not exploited by Re-Pair, unless its translates into similar adjacency lists. This, however, makes our technique independent of the numbering. In the work of [BV04a] 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 lexicographic order, 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 (in contrast, [BC08] use ζ-coding to reduce space). 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 due to the need of adding up differences. Figure 3 shows the differential encoding of the graph shown in Figure 1.

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

T(G) =	-1	1	3	4	-2	1	3	4	-3	4	5	-4	3	4	5	6	-5	-6	4	5
Diffs(T(G)) =	-1	1	2	1	-2	1	2	1	-3	4	1	-4	3	1	1	1	-5	-6	4	1

Figure 3: Differential encoding of the graph shown in Figure 1.

$1, 2, 4, 5$. 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, so we cannot combine this method with differences. Decompression and traversal, on the other hand, are not affected at all. The experimental results show that this approach achieves better compression ratios than applying Re-Pair without differences. Note that this reordering is just a heuristic, and one could aim to finding the optimal ordering. However, similar problems have been studied for differential encoding of inverted lists, and they have been found to be hard [FV99, SCSC03]. Indeed, the whole point of the work of [BC08] is to develop heuristics to find good subsequences efficiently.

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. See Figure 2 for an example on the values assigned in B_1 and B_2.

5 Lempel-Ziv Compression of Web Graphs

The Lempel-Ziv compression family [ZL77, ZL78] achieves compression by replacing repeated sequences found in the text by a pointer to a previous occurrence thereof. In particular, the LZ78 variant [ZL78] 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 [BV04a], `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. For larger graphs 4 bytes per node id would not suffice, and we would spend even more space per edge in a plain representation. Later on we introduce a larger graph to study the scalability of our approach.

Crawl	Nodes	Edges	Edges/Nodes	Plain size (MB)
`EU (2005)`	862,664	19,235,140	22.30	77
`Indochina (2004)`	7,414,866	194,109,311	26.18	769
`UK (2002)`	18,520,486	298,113,762	16.10	1,208
`Arabic (2005)`	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 load 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 of 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 [LM00] uses at the very least 200% extra space). The rest of our experiments are run using 3% extra space[5].

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 [GN07] that requires 260 MB total space for `EU` and 2.4 GB 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.

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

`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 4 GB of RAM for the larger graphs.

It is interesting to notice that, as k doubles, compression time is almost halved (especially for small k). This is related to the fact that we cannot 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 $k = 10,000$.

keep using the absolute space originally allowed.

`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.

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 4 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, that just 15 MB of RAM is needed for our largest graph, `Arabic`. If we extrapolate to the 4 TB graph of the *whole* indexed Web mentioned in the Introduction, we get that we could handle it in secondary memory while using 20–45 GB of RAM (64 GB RAM servers are becoming commonplace). If the

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

compression would stay at about 3 bpe (as in our largest graph, Section 6.5), this would mean that access to the compressed Web graph would be up to 10 times faster than in uncompressed form, on disk.

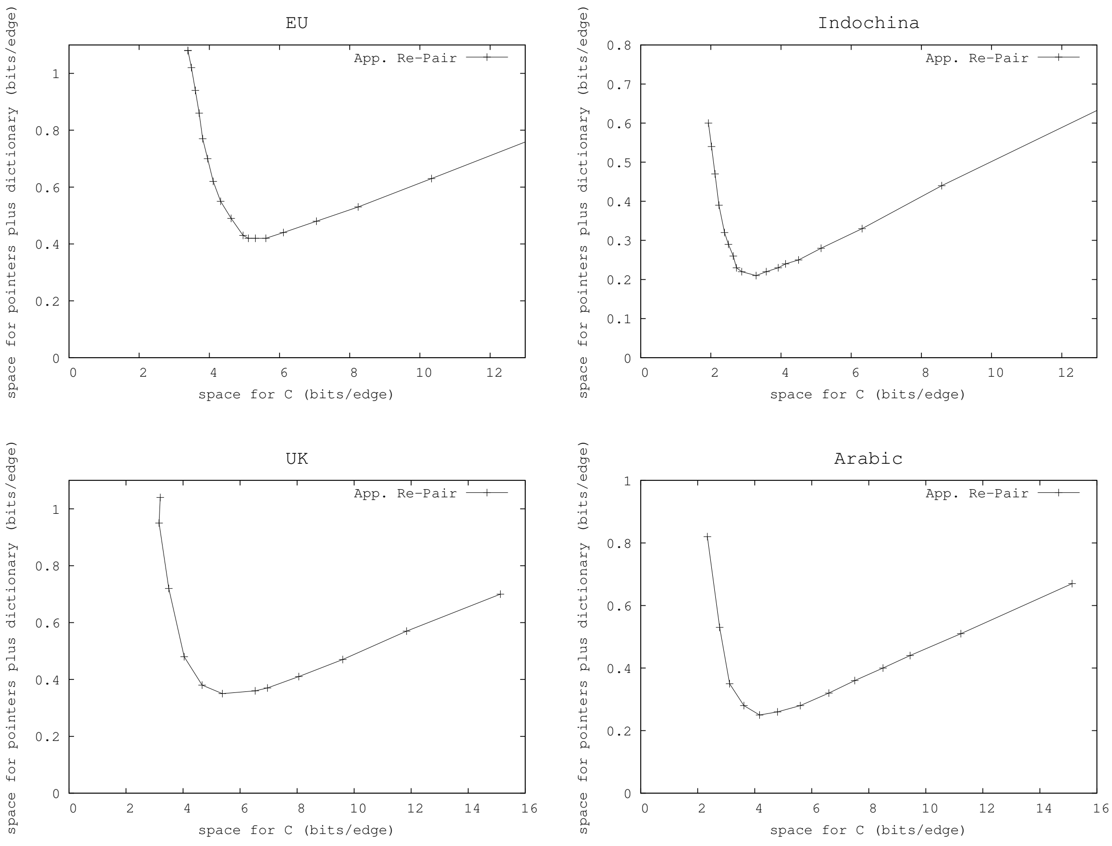

Figure 4: Space used by the sequence versus the dictionary plus the pointers, all measured in bits per edge.

6.3 Compressed Graph 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).

- *Re-Pair*: Normal Re-Pair.
- *Re-Pair Diffs*: Re-Pair with differential encoding.

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

- *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 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. This is a sort of worst-case situation for real traversals, which might exhibit some locality; we explore DFS and BFS traversals in Section 6.5.

These experiments were run on a Pentium IV 3.0 GHz with 4 GB of RAM using Ubuntu GNU/Linux 8.10 with kernel 2.6.27-16 and `g++` with `-O9` and `-DNDEBUG` options.

We compared to the implementation by [BV04a], run on our machine, with various space/time tradeoffs. The implementation of Boldi and Vigna gives a size measure that is consistent with the sizes of the generated files (and with their paper [BV04a]). This is the space we report, despite the process (in Java[8]) actually needs more memory to run. 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[9].

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.

Figure 5 shows 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

[8]Using the Sun Java virtual machine provided by Ubuntu 8.10 default packaging system, v1.6.0_14.

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

of them. The time/space tradeoff between compacting the dictionary or not should be almost the same for the other Re-Pair implementations too.

Two lines display the best current tradeoffs offered within the *WebGraph* project, which is the state of the art for Web graphs. These are labeled BV, which is the current release in their site (version 2.4.3), and BSV [BSV09], a variant that uses another node ordering, called `shbhGray`, to achieve better tradeoffs (available within the same release).

From a pure compression standpoint, our results are not competitive with those of *WebGraph* (except on the small `EU` graph). However, when we consider the time to access the graphs, it turns out that *WebGraph* needs significantly more space than the minimum in order to provide reasonable navigation times. In particular, it is still 1.5–2 times slower than our representations when using the same amount of space. 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 plain version and 4–8 times slower.

No traversal times are reported by [BC08] for their "virtual node mining" (VNM) approach. However, we can distinguish two stages in VNM compression: (1) identifying bi-cliques and replacing them by virtual nodes; (2) ζ-encoding the resulting adjacency lists. Our results in Figure 5 indicate that our extraction is currently so fast that even the use of differential encoding (that is, we have to add the current number to the previous one before reporting it) makes a noticeable difference in performance. Thus decoding a ζ-encoded bitstream is likely to make the VNM approach considerably slower than ours. Our encoding is just a plain sequence of integers (all using a fixed number of bits). On the other hand, if VNM omitted stage (2) and used a plain sequence of integers, the access time of both approaches should be very similar, as both have to (recursively) expand virtual nodes (or nonterminals, in our case) until obtaining the final list of real nodes (terminals, in our case).

Therefore, we divide the comparison into two parts. In this section we test how would both methods compare in space if VNM omitted stage (2) in order to reach the speed of our method. In Section 6.4 we consider how to achieve further space reduction in our method and compare with the full VNM approach.

Table 5 shows the compression ratios of the methods measured in number of integers of the adjacency lists before and after grammar compression. For VNM we use the data [BC08] give in their Figure 3 (left), which accounts for all the edges in the graph before and after replacing bi-cliques by virtual nodes. For our method, we present a plain approach (*Plain*) where we add the length (in integers) of sequence C (the adjacency list after compression) plus two integers per nonterminal, which corresponds to a plain representation of the dictionary R. Interpreted in terms of virtual nodes, this corresponds to the fact that our "virtual nodes" (nonterminals) have always outdegree 2. We also present the more realistic approach where we add the length of C and the integers in our compressed representation of dictionary R. Again interpreted in terms of virtual nodes, this acknowledges the fact that, despite we use Re-Pair, which creates nonterminals as pairs of other symbols, one could unroll part of the recursion and rewrite nonterminals as sequences of nodes (this is what is done, implicitly, in the dictionary compression technique of [GN07]). Furthermore, we show versions *Diff* and *Reord*. Note the former deviates from the model of virtual nodes (as its nonterminals are differences rather than node identifiers, and this has no metaphor in terms of graph nodes) and the latter is able of detecting noncontiguous subsequences thanks to

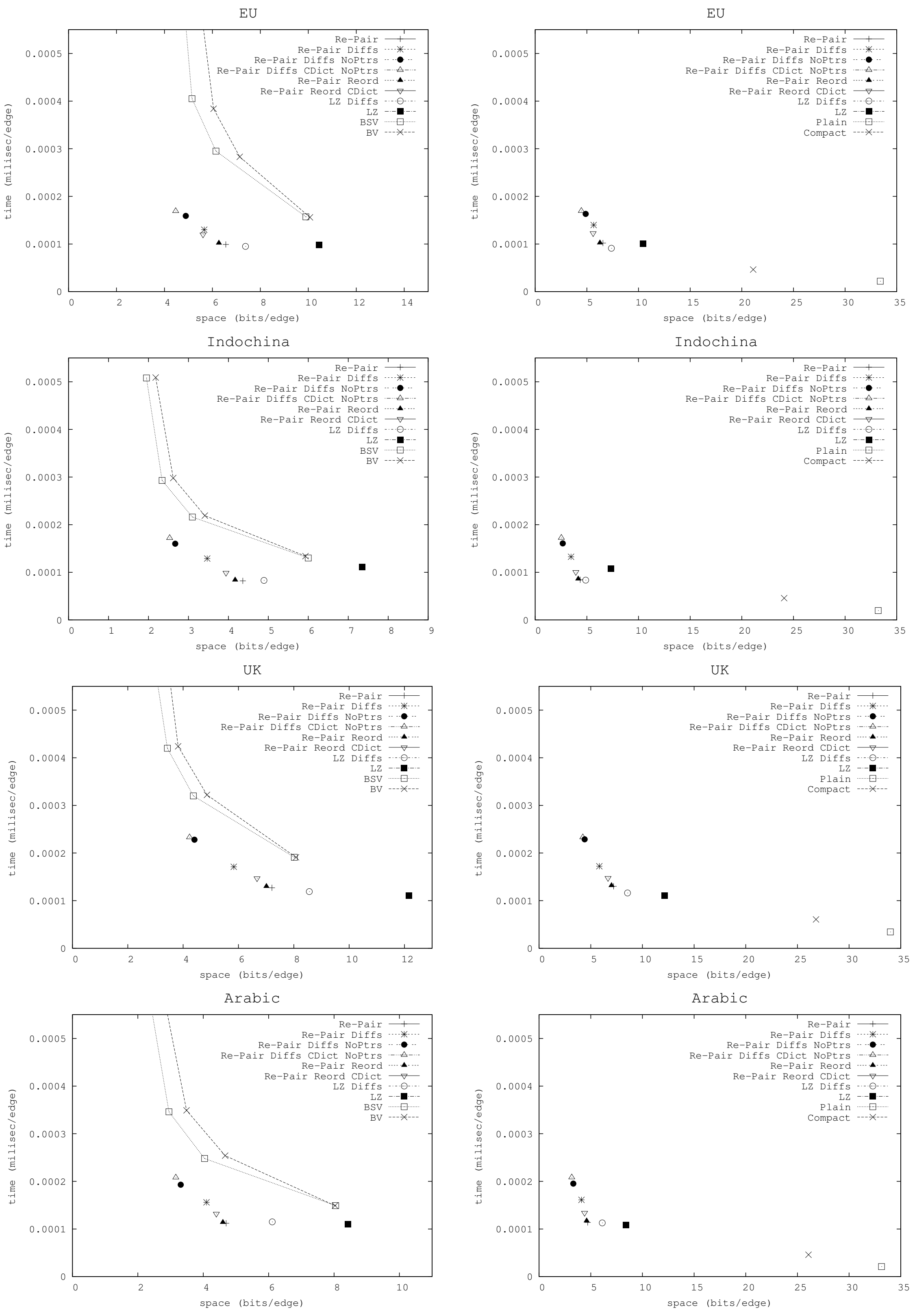

Figure 5: Space and time to find neighbors for different graph representations, over the four crawls.

the reordering. Note that in all cases we are ignoring the cost of the vector of pointers from nodes to adjacency lists.

Graph	Ours Plain Diff	Ours Plain Reord	Ours Diff	Ours Reord	VNM
`EU`	21%	24%	20%	20%	22%
`Indochina`	11%	14%	10%	12%	
`UK`	17%	22%	15%	19%	20%
`Arabic`	13%	15%	11%	13%	14%

Table 5: Compression ratios (in percentage) achieved by different grammar compression methods, before any further encoding of the sequences.

From the table we can conclude that, if we consider the variant of our grammar compression that can most easily be identified as a mechanism to add virtual nodes to the graph in order to factor out edges, that is, our *Plain Reord* variant, the result is 7% to 10% worse than VNM. Note that *Reord* does capture common subsequences as well, but VNM does it better. However, our model is not exactly like that of creating virtual nodes, and so the comparison is not only "who finds better virtual nodes". Our non-*Plain* variants take advantage of the hierarchical structure of our nonterminals to represent them using fewer integers, and our *Reord* variant already improves upon VNM by 5% to 14%. Furthermore, *Diffs* totally deviates from the virtual nodes model, as explained. In non-*Plain* form, it takes 9% to 25% less space than VNM, yet it could be slightly slower due to the need of adding up the differences.

To summarize: If we take only the grammar-compression aspect of VNM, and encode its adjacency lists as a sequence of integers using a fixed number of bits, both techniques are likely to achieve similar speeds, yet ours would take up to 14% less space (if we consider variant *Reord*, as *Diff* is slightly slower). Actually, VNM produces fewer virtual nodes than we produce nonterminals, and thus their integers use potentially fewer bits. In the tested graphs, however, this only makes a difference of 2% on `Arabic`, yet *Reord* is still 4% smaller.

As explained, VNM achieves further space reduction via ζ-coding. In the next section we consider how we can also our space at the expense of higher access time, and up to which point can we match the space achieved by VNM.

6.4 Further Compression

Our experiments indicate that our technique offers a good space/time tradeoff, yet it is unable to achieve the best compression ratios reached by alternative methods [BSV08, BSV09, BC08]. We explore now how far can we reach in terms of compression ratio, even if sacrificing access time.

As explained, compressed sequence C is stored with fixed-length integers, and possibly amenable of further compression. The techniques used in other schemes, such as ζ-encoding the differences, do not work well after Re-Pair factors out common pairs of differences ((1,1), (1,2), etc.). Nevertheless, it turns out that the zero-order entropy of C is low enough to permit compression: After applying Re-Pair, every *pair* of symbols in C is unique, yet *individual symbols* are not.

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 that of [HT71], 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 ($n \le \sigma \le n + |R|$), 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: $\mathit{Huffman} \ge \sigma \log \sigma + \sum_{i \in \Sigma} n_i \log \frac{n}{n_i}$, 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: $2\sigma + \sum_{i \in \Sigma} n_i \log \frac{n}{n_i} \le HT \le 2\sigma + \sum_{i \in \Sigma} n_i \left(\log \frac{n}{n_i} + 2\right)$, 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 6 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[10]. Table 7 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 6: 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 7: 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.

[10] 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.

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 wavelet tree implementation [CN08] 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 8 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 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.

Graph	Orig.	WT(8)	WT(32)	WT(128)	WT(∞)	BV(∞)	VNM	VNM(∞)
`EU`	4.47	4.59	3.71	3.49	3.42	4.38	4.07	2.90
`Indochina`	2.53	2.52	1.97	1.84	1.79	1.47		
`UK`	4.23	4.36	3.40	3.16	3.08	1.70	3.75	1.95
`Arabic`	3.16	3.34	2.60	2.42	2.36	1.99	2.91	1.81

Table 8: Total space, measured in bpe, achieved when using compressed wavelet trees to represent C, with different sampling rates. We also show the best results of *WebGraph*, and those of virtual node mining, with and without direct access.

The 7th column of Table 8 shows the best result reported in the *WebGraph* site. It beats our best results except on `EU`, but that representation does not support direct access. The last two columns of Table 8 show the results reported by [BC08], first adding (VNM) and second not adding (VNM(∞)) the space of their pointers array. The compression ratios of VNM(∞) are also unreachable for us, but again this variant does not provide direct access. The ratios VNM achieves with direct access, instead, are between our WT(8) and WT(32). It is likely that VNM, even with ζ-codes, will be faster than our wavelet-tree-based variants. Tradeoffs between VNM and VNM(∞) can be achieved by sampling the array of pointers (for example, storing some absolute values and then differentially encoding the subsequent ones). Moreover, one could consider combinations such as compressing the VNM adjacency lists with wavelet trees, for example, and then use our *NoPtrs* variants.

Considering the results of the previous section, we note that the merit of the VNM approach is not that it finds a smaller grammar than ours (it usually does not, as shown), but in that its resulting sequence seems to be more compressible. On the other hand, we have proposed compression techniques (like using wavelet trees) that do not affect the width of the pointers, and thus could offer different space/time tradeoffs to VNM.

6.5 Scalability

A disadvantage of our method is that Re-Pair compression is offline, thus we cannot process the graph incrementally as for example [BV04a]. In case of a large graph we can resort to secondary memory as described in Section 3.3, yet compression will require multiple passes on disk (recall Table 4). In this section we explore an alternative solution, which is also relevant for a distributed processing scenario. We propose a simple heuristic that exploits the locality of reference: We partition the graph into pieces of the maximum size we can handle in main memory and compress them separately, considering them as independent graphs. We will call each part of the graph a *subgraph*, even when formally they are not, as some of them point to nodes that are not in the subgraph.

Our compression method does not require the node identifiers to be in a given range. Rather, it regards the lists as generic sequences, so it does not matter if a node in a subgraph points to a non-existing identifier. Therefore, we do not rename the nodes in the adjacency lists inside each subgraph. As a consequence, when we obtain the adjacency list of a node, no mapping is required.

Our final graph representation is an array of subgraphs represented using our technique and an array of offsets, `offs`, containing the absolute node identifier of the first adjacency list represented in each subgraph. For retrieving the neighbors of a node v, we search for the largest index i such that $\texttt{offs}[i] \leq v$. Then we retrieve the $(v - \texttt{offs}[i] + 1)$-th list of the i-th subgraph.

We tested this approach on the `uk-union-2006-06-2007-05` graph (`uk-union` from now on), the largest crawl available at the *WebGraph* site. It has $133,633,040$ nodes and $5,507,679,822$ edges (41.21 edges/node). Its plain adjacency list representation requires 22 GB of memory.

We partitioned the graph into 9 pieces. The first eight were cut just before passing the 650M edges barrier, and the last was the remainder. The first eight pieces are 2.5 GB in size, the last one is 1.2 GB. We compressed the graph using *Re-Pair Diffs CDict NoPtrs*, our smallest practical variant.

We achieved 2.91 bpe for `uk-union` using $k = 100,000$ and 3% extra space on top of each sequence when compressing. The final result requires just less than 2 GB for operating the whole graph, which can be handled in main memory by most commodity PCs.

We compare our technique with BV, which performed best on our previous experiments. This time, instead of giving the average time to retrieve each neighbor from a random node, we opt for demonstrating the performance when carrying out the two most typical graph traversals: depth-first-search (DFS) and breadth-first-search (BFS).

Even when our implementation was perfectly capable of running both algorithms in the original machine using less than 3 GB[11], the overhead imposed by the Java virtual machine on B(S)V made it impossible to execute the process within that space. So for this experiment we switched to an Intel(R) Xeon(R) CPU running at 2 GHz, with 8 cores and 16 GB of RAM, running Ubuntu GNU/Linux (Server) with kernel 2.6.24-27 in 64-bit mode. For both traversals we implemented a similar queue/stack [CLRS01] using arrays, to make sure that the STL and the Java API were not altering the performance results.

Table 9 shows the time for the two traversals, including the space required by each representation, and displaying two tradeoff points for BV. The situation is as for all previous experiments:

[11]That is, the memory limit for a process on the GNU/Linux kernel on 32-bit machines.

Method	bpe	DFS(sec)	BFS(sec)
Ours	2.91	632	636
BV	2.58	1,194	1,168
BV	3.22	740	722

Table 9: Time and space tradeoffs obtained for `uk-union` when running BFS and DFS traversals.

BV is able of achieving less space than our representation, but ours is faster when both use the same amount of space (or BV uses even more, as in this case).

It is interesting to note that, using the wavelet tree, we achieve as little as 2.17 bpe, less than the best space reported in *WebGraph*. Still retaining the current access times, our representation could still achieve better space by choosing a smaller k, as shown in Table 3. Another interesting point to mention is that B(S)V include some caches in their code to speed up recently decompressed lists. This is quite relevant for the traversals, given the locality of references and the fact that, in their method, nearby lists have to be decompressed to obtain the desired list. We could aim at doing something similar with frequent nonterminals, yet this requires serious further study as it has a price in terms of extra memory.

Finally, we remark that, since our representations work with any node ordering, we could use orderings suitable for running external memory algorihtms [Vit06] so that our reduction of space on disk would translate into reduced execution times, in case the compressed graph does not fit in RAM. We could also refine our partitioning technique by using separators [BBK03] to reduce the amount of external links, which may miss some compression opportunities, or use techniques to merge the dictionaries of the different partitions [Wan03].

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 [LM00] and LZ78 [ZL78]. 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, although unable to match the compression ratios of *WebGraph* [BV04a, BSV08, BSV09] (the state of the art), is 1.5–2 times faster to navigate the compressed graph when both structures are given the same amount of space to operate. 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 our representation 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 in main memory even in compressed form, our scheme adapts well to secondary memory, where it can make fewer accesses to disk and/or shorter seeks 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.

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.

Recent work [BC08] confirms that grammar-based compression is indeed an extremely promising avenue for future research. Unlike our representation, theirs achieve better compression ratios than *WebGraph*. Although their access time is not clear, our limited experiments show that the techniques achieve comparable space/time tradeoffs and, more importantly, that some techniques could be combined to achieve an improved representation.

Another line of research focuses on adding more functionality to the compact representations, further than retrieving the neighbors of a node. For example, some sampling algorithms on the Web [KKR+99, Kle99] require access to the nodes pointing to the current one, that is, reverse navigation. There has been some progress on providing bidirectional navigation, within space that is more than those shown in this paper for simple forward navigation, but less than that of representing the original and the transposed graph [BKM+00, BLN09, CN10]. Their main problem is that, due to their non-local access pattern, they succeed only if the graph fits in main memory.

Acknowledgements

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

References

[ACL00] 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.

[AM01] M. Adler and M. Mitzenmacher. Towards compressing Web graphs. In *Proc. 11th Data Compression Conference (DCC)*, pages 203–212, 2001.

[AMN08] Y. Asano, Y. Miyawaki, and T. Nishizeki. Efficient compression of Web graphs. In *Proc. 14th Conference on Computing and Combinatorics (COCOON)*, LNCS 5092, pages 1–11, 2008.

[BBH+98] 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.

[BBK03] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. 14th Symposium on Discrete Algorithms (SODA)*, pages 579–588, 2003.

[BBYRNZ01] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proc. 8th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 10–20, 2001.

[BC08] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to Web graph compression with communities. In *Proc. International Conference on Web Search and Web Data (WSDM)*, pages 95–106, 2008.

[BKM$^+$00] 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.

[Bla06] 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.

[BLN09] N. Brisaboa, S. Ladra, and G. Navarro. K2-trees for compact Web graph representation. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 18–30. Springer, 2009.

[BSV08] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008.

[BSV09] P. Boldi, M. Santini, and S. Vigna. Permuting Web graphs. In *Proc. 6th Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 116–126, 2009.

[BV04a] P. Boldi and S. Vigna. The WebGraph framework I: compression techniques. In *Proc. 13th World Wide Web Conference (WWW)*, pages 595–602, 2004.

[BV04b] P. Boldi and S. Vigna. The WebGraph framework II: Codes for the world-wide web. In *Proc. 14th Data Compression Conference (DCC)*, page 528, 2004.

[CGH$^+$98] 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 1443*, pages 118–129, 1998.

[Cla96] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

[CLL$^+$05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2nd edition, 2001.

[CN07] F. Claude and G. Navarro. A fast and compact Web graph representation. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 105–116, 2007.

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[CN10] Francisco Claude and Gonzalo Navarro. Extended compact web graph representations. In Tapio Elomaa, Heikki Mannila, and Pekka Orponen, editors, *Algorithms and Applications*, volume 6060 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.

[CPMF04] 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.

[DL98] N. Deo and B. Litow. A structural approach to graph compression. In *Proc. of the 23th MFCS Workshop on Communications*, pages 91–101, 1998.

[DLL$^+$06] D. Donato, L. Laura, S. Leonardi, U. Meyer, S. Millozzi, and J.F. Sibeyn. Algorithms and experiments for the Webgraph. *Journal of Graph Algorithms and Applications*, 10(2):219–236, 2006.

[FMMN07] 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.

[FV99] A. Fink and S. Voß. Applications of modern heuristic search methods to pattern sequencing problems. *Computers & Operations Research*, 26:17–34, 1999.

[GGV03] 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.

[GMR06] 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.

[GN07] 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.

[HKL00] 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.

[HT71] T. Hu and A. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM Journal of Applied Mathematics*, 21:514–532, 1971.

[Jac89] G. Jacobson. *Succinct Static Data Structures.* PhD thesis, Carnegie Mellon University, 1989.

[KKR$^+$99] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The Web as a graph: Measurements, models, and methods. In *Proc. 5th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 1627, pages 1–17, 1999.

[Kle99] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[KRRT99] 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.

[LM00] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[MR97] 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.

[Mun96] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[RGM03] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proc. 19th International Conference on Data Engineering (ICDE)*, page 405, 2003.

[RRR02] 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.

[RSWW01] 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.

[Ryt03] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[Sak05] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2-4):416–430, 2005.

[SCSC03] W. Shieh, T. Chen, J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Information Processing & Management*, 39(1):117–131, 2003.

[STKA07] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms. In *Proc. 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*. ACM Press, 2007.

[SY01] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. 11th Data Compression Conference (DCC)*, pages 213–222, 2001.

[TGM93] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proc. 2nd International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 8–17, 1993.

[Vit06] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[Wan03] R. Wan. *Browsing and Searching Compressed Documents.* PhD thesis, Dept. of Computer Science and Software Engineering, University of Melbourne, 2003.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23:337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.