Compressing Web Graphs like Texts *

Gonzalo Navarro

Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl

Abstract

The need to run different kinds of algorithms over large Web graphs motivates the research for compressed graph representations that permit accessing without decompressing them. At this point there exist a few such compression proposals, some of them very effective in practice. In this paper we introduce a novel approach to graph compression, based on regarding the graph as a text and using existing techniques for text compression/indexing. This permits accessing the graph efficiently without decompressing it, and in addition brings in new functionalities over the compressed graph. Our experimental results show that our technique has the potential of being competitive with the best alternative techniques.

1 Introduction and Related Work

Compression techniques aim at reducing the space requirement to store data. A recent exciting research area is that of *compressed data structures*. A compressed data structure uses little space, yet it can carry out the same operations of its classical (uncompressed) counterpart without the need to uncompress it. Compressed data structures have been proposed to handle sets and sequences [Jac89, Mun96, Cla96, Pag99, RRR02], trees [MR97, GRRR04], planar graphs [Jac89, MR97], permutations [MRRR03], texts [Sad03, Nav04, FM05, GV06, MN05, FMMN06], and others.

Compressed data structures are usually slower than their uncompressed counterpart, yet they have the attractive of permitting manipulation in main memory of large objects, which would need secondary memory storage if represented in uncompressed form. Secondary memory is so slow in comparison to primary memory that a significantly slower compressed data structure may pay off in comparison to a disk access.

In this scenario, graphs are in the curious position of being among the structures receiving most attention from the algorithmic community, and at the same time not much attention from the compression point of view, especially regarding compressed data structures for graphs. Even the existing definitions of graph entropy are relatively naive and do not account for the most interesting types of regularities that appear, for example, in Web graphs.

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

*This work has been funded by a grant from Yahoo! Research Latin America.

The existing work on Web graph compression mainly focuses on several heuristic observations over those graphs, such as their skewed indegree and outdegree distributions, repetitiveness in the sets of outgoing links, locality in the references, and so on. Probably the best result in practice is that of Boldi and Vigna [BV03], where they achieve about 6 bits/link and permit access to incoming and outgoing links of any node within the microsecond per delivered link.

In this paper we propose to regard the graph as a collection of texts, so as to use the wealth of existing machinery for compressed text self-indexing. A self-index is a data structure that provides indexed access to a text and at the same time can reproduce any substring of it, thereby replacing the text. We show how some existing compressed text indexes can be adapted to compress graphs and how the graph operations can be mapped to operations on the induced text collection. A nice feature of using self-indexes is that outgoing links are retrieved by an operation similar to the text extraction functionality, whereas incoming links are retrieved by a search operation over the self-index. Thus, self-indexes give in a single data structure the forward and reverse link retrieval functionality, which in several alternative schemes must be implemented by indexing the original and the transposed graph.

Finally, we give experimental results for one of the best indexing alternatives. We compare against the best practical result we are aware of [BV03] and show that our technique is competitive. Both techniques achieve a space/time tradeoff. We show that, although our technique cannot reach space occupancies as low as that of Boldi and Vigna (as we can hardly take less than 7 bits/link), it can be equally fast on some operations when slightly more space (8–10 bits/link) is given to both. This shows that our approach, besides being an interesting line of attack and fun by itself, is promising in practice.

2 Related Work

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

The incidence matrix permits determining in constant time whether two nodes are neighbors or not, but listing the neighbors or reverse neighbors of a node requires $O(n)$ time. The adjacency list requires $O(n)$ time to determine neighborhood, but each neighbor of a node can be generated in constant time (other $e \log n$ bits are required to achieve the same for reverse neighbors). Adjacency lists can achieve constant time to determine neighborhood by replacing the lists with perfect hash tables, within $O(e \log n)$ bits of space.

The oldest work on graph compression focuses on undirected unlabeled graphs. The first result we know of [Tur84] shows that planar graphs can be compressed into $O(n)$ bits. The constant factor was later improved [KW95], and finally a technique yielding the optimal constant factor was devised [HKL00]. Results on planar graphs can be generalized to graphs with constant *genus* [Lu02]. More generally, a graph with genus g can be compressed into $O(n)$ bits [DL98]. The same

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

holds for a graph with g *pages*. A page is a subgraph whose nodes can be written in a linear layout so that its edges do not cross. Hence edges of a page form a nested structure that can be represented as a balanced sequence of parentheses.

Some classes of planar graphs have also received special attention, for example trees, triangulated meshes, triconnected planar graphs, and others [IR82, KW95, HKL99, Ros99]. For dense graphs, it is shown that little can be done to improve the space required by the adjacency matrix [Nao90].

The above techniques consider just the compression of the graph, not its access in compressed form. The first compressed data structure for graphs we know of [Jac89] requires $O(gn)$ bits of space for a g-page graph. The neighbors of a node can be retrieved in $O(\log n)$ time each (plus an extra $O(g)$ complexity for the whole query). The main idea is again to represent the nested edges using parentheses, and the operations are supported using succinct data structures that permit navigating a sequence of balanced parentheses. The retrieval was later improved to constant time by using improved parentheses representations [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.

All those techniques based on number of pages are unlikely to scale well to more general graphs, in particular the Web graph. 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 suporting access to the compressed graph), the most striking 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.

They [Bla06] apply the separator technique to design a compressed data structure that gives constant access time per delivered neighbor. Reverse neighbors must be handled by using another index on the reversed edges. They carefully implement their techniques and experiment on several graphs. In particular, on a graph of 1 million (1M) nodes and 5M links from the Google programming contest[2], their less space-demanding data structure (built for both normal and reverse edges) requires 13.39 bits per edge (bpe), and it is about 5 times slower than a plain uncompressed representation using arrays for the adjacency lists. The authors favor a slightly larger representation requiring 16.04 bpe, which is 4 times slower than the plain representation. It is not clear how these results would scale to larger graphs, as much of their improvement relies on smart caching, and this effect could vanish with real Web graphs.

There is also some work specifically aimed at compression of Web graphs [BKM$^+$00, AM01, SY01, BV03]. In this graph the (labeled) nodes are Web pages and the (directed) edges are the hyperlinks. Several properties of Web graphs have been identified and exploited to achieve compression:

Skewed distribution: The indegrees and outdegrees of the nodes distribute according to a power law, that is, the probability that a page has i links is $1/i^\theta$ for some parameter $\theta > 1$. Several experiments give reasonably consistent values of $\theta = 2.1$ for incoming links and $\theta = 2.72$ for outgoing links [ACL00, BKM$^+$00].

[2] `www.google.com/programming-contest`

Locality of reference: Most of the links from a site point within the site. This motivates in [BBH+98] the use of lexicographical URL order to list the pages, so that outgoing links go to nodes whose position is close to that of the current node. Gap encoding techniques are then used to encode the differences among consecutive target node positions.

Similarity of adjacency lists: Nodes close in URL lexicographical order share many outgoing links [KRRT99, BV03]. This permits compressing them by a reference to the similar list plus a list of edits. Moreover, this translates into source nodes pointing to a given target node forming long intervals of consecutive numbers, which again permits easy compression.

In [SY01] they partition the adjacency lists considering popularity of the nodes, and use different codings method for each partition. A more hierarchical view of the nodes is exploited in [RGM03]. In [AM01, RSWW01] they take explicit advantage of the similarity property. A page with similar outgoing links is identified with some heuristic, and then the current page is expressed as a reference to the similar page plus some edit information to list the deletions and insertions neede to obtain the current page from the referenced one. Finally, probably the best current result is from [BV03], who build on previous work [AM01, RSWW01] and further engineer the compression to exploit the properties above.

Experimental figures are not always easy to compare, but they give a reasonable idea of the practical performance. Over a graph with 115M nodes and 1.47 billion (1.47G) edges from the Internet Archive, [SY01] require 13.92 bpe (plus around 50 bits per node), yet they cannot retrieve reverse neighbors. In [RSWW01], over a graph of 61M nodes and 1G edges, they achieve 5.07 bpe for the graph and 5.63 for its transpose (so they require both, 10.7 bpe, to access neighbors and reverse neighbors). In [AM01] they achieve 8.3 bpe (no information on bpn (bits per node)) over TREC-8 Web track graphs (WT2g set), yet they cannot access the graph in compressed form (and thus do not need to care for reverse neighbors, in particular). In [BKM+00] they require 80 bits per node plus 27.2 bpe (and can answer reverse neighbor queries as well).

By far the best figures are from [BV03]. For example, they achieve space close to 3 bpe to compress a graph of 118M nodes and 1G link from WebBase[3]. This space, however, is not sufficient to access the graph in compressed form. An experiment where the required extra information on the graph and its transpose is stored is carried out on a graph of 18.5M nodes and 292M links, where they need 6.89 bpe to achieve access times below the microsecond. Those access times are of the same order of magnitude of other representations [SY01, RGM03, RSWW01]. For example, the latter reports times around 300 nanosecods per delivered edge.

3 Compressed Text Indexing

Let us now consider a text $T = t_1 \ldots t_n$, which is a sequence of n symbols over an alphabet Σ of size $|\Sigma| = \sigma$. Compressed data structures for text have been intensively studied in this decade [Sad03, Nav04, FM05, GV06, MN05, FMMN06]. The research has evolved into the concept of *self-indexing*. A *compressed full-text self-index* is a data structure over a text T providing the following functionality:

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

1. Given a pattern $P = p_1 \ldots p_m$ over Σ, *count* the number of occurrences of P in T, that is, tell how many times does P occur in T.

2. Given P, *locate* the occurrences of P in T, that is, find the positions where P occurs in T.

3. Given a pair of positions l, r in T, *display* the content of $T_{l,r} = t_l \ldots t_r$.

The latter operation permits us to access any substring of T (including the whole T itself), thus making it unnecessary to store T. Thus a self-index replaces the text and at the same time provides some search operations on it. In addition, a compressed self-index requires space proportional to the size of the compressed text.

We review now one of those compressed self-indexes, the *Compressed Suffix Array (CSA)* of Sadakane [Sad03], which we use as the basis for our compressed graph representation. As seen later, our graph representation will induce a large alphabet, and the CSA is especially well suited to large alphabets.

The *suffix array* [MM93] of T is a permutation A of $[1, n]$ such that $T_{A[i],n}$ is lexicographically smaller than $T_{A[i+1],n}$ for all $1 \leq i < n$. For those comparisons to be well defined let us assume that T finishes with a special character $t_n = \$ \in \Sigma$, lexicographically smaller than any other character in Σ and not appearing elsewhere in T. Those strings of the form $T_{i,n}$ are called *suffixes* of T, and A points to all the suffixes of T in lexicographical order. As every substring of T is the prefix of some suffix, it follows that all the occurrences of P in T are pointed from a contiguous range in A, as the suffixes prefixed by P form a lexicographic interval. This range, $A[sp, ep]$ can be binary searched by comparing P against strings $T_{A[i],A[i]+m-1}$, in overall time $O(m \log n)$.

The CSA replaces A by several more compact structures. The most important is Ψ, a permutation of $[1, n]$ defined so that $A[\Psi(i)] = A[i] + 1$. To replace A we also need C, an array over Σ so that $C[a]$ is the number of occurrences of characters smaller than a in T. Although we cannot recover A from Ψ and C, we can simulate the binary search in C. More precisely, we are able to extract the substrings $T_{A[i],A[i]+m-1}$ required by the binary search, as follows. As the first characters of the suffixes pointed from A are ordered, $T_{A[i]}$ must be the character a such that $C[a] < i \leq C[a + 1]$. This is easily discovered by a binary search. To obtain the next character of the suffix, $T_{A[i]+1}$, we should know where is that text position pointed to from A, that is, for which i' it holds $A[i'] = A[i] + 1$, so as to repeat the argument above with i'. This is where Ψ comes into play, as $i' = \Psi(i)$. Therefore, with C and Ψ we are able to find sp and ep.

To avoid the binary searches in C we can replace it with a bit vector $D[1, n]$, so that $D[1+C[a]] = 1$ for $a \in \Sigma$ and 0 elsewhere; and a string S of length at most σ containing the different characters that appear in T in lexicographic order. Then the character corresponding to $T_{A[i]}$ is $S[rank(D, i)]$, where $rank(D, i)$ is the number of bits set in $D[1, i]$. This $rank$ query can be solved in constant time by storing $o(n)$ bits in addition to D [Jac89, Mun96, Cla96], and even within $\sigma \log n + o(n)$ bits of space without explicitly storing D, as D has at most σ bits set [RRR02]. The complementary query, used later in the paper, is $select(D, j)$, which gives the position of the j-th bit set in D. It can also be solved in constant time within the same space bounds [Mun96, Cla96, RRR02].

Thus, assuming we have constant-time access to Ψ, we can solve a counting query (returning $ep - sp + 1$) in time $O(m \log n)$ by storing Ψ, D, and S. Yet, in principle Ψ is a permutation that requires as much space as A. However, Ψ is compressible to a size that depends on the empirical entropy of T [Sad03, MN05]. Some known properties of Ψ are: (1) Ψ is increasing within the

suffixes that start with the same character; (2) if we represent those differences between consecutive increasing values using δ-coding [Eli75], the total number of bits required is $nH_0(T)+O(n \log \log \sigma)$; (3) if we call a *run* in Ψ a maximal area $\Psi(x, y)$ where $\Psi(i+1) = \Psi(i)+1$ for all $x \leq i < y$, then Ψ can be covered with at most $nH_k(T) + \sigma^k$ runs, for any k, and thus it can be represented using this number of entries. Informally, frequently repeated substrings in T translates into long runs in Ψ. Sadakane takes advantage of property (2) so as to achieve $nH_0 + O(n \log \log \sigma)$ bits to represent the whole index. This compressed representation can still be accessed in constant time using Four-Russians techniques and some additional tables, see [Sad03] for details.

For locating the occurrences, it is necessary to store some *samples* of A. T is sampled at regular intervals d and the values i such that $T_{j,n}$ is pointed from $A[i] = j$ are stored. More precisely, we store the pairs $(j, A^{-1}[j]) = (A[i], i)$ for all $j = A[i]$ multiple of d. In order to locate the occurrence $A[i]$ for some $sp \leq i \leq ep$ (recall that we are not storing A), we first check whether i has been sampled. If it is sampled, that is, $(A[i], i)$ is stored, we deliver the value $A[i]$. Otherwise we try with $i' = \Psi(i)$, $i'' = \Psi(\Psi(i))$, and so on, until we find a sampled value $(A[i^*], i^*)$ after s steps. Since $A[\Psi(i)] = A[i]+1$, it follows that our original value was $A[i] = A[i^*] - s$. Moreover, the number of steps will be at most the sample size because we are virtually traversing T in forward direction until finding a sampled text position. To know in constant time whether a position i is sampled, we can use a bit array $Smp[1, n]$ so that $Smp[i] = 1$ iff i is sampled, plus an array Pos containing the values $A[i]$, that is, $Pos[rank(Smp, i)] = A[i]$. Thus the process takes worst-case time proportional to the sample size d. If this is $O(\log n)$, we have $O(\log n)$ time to locate each occurrence and $O(n)$ extra bits to store the samples.

Finally, to display a text substring $T_{l,r}$, we already know how to obtain the consecutive characters of $T_{A[i],n}$, so we only need to find $i = A^{-1}[l]$. The sampling used for locating is useful again. Assumed $APos$ stores the values i we have sampled in Smp, stored by increasing text positions $j = A[i]$. Then the text sample preceding l is $p = \lfloor l/d \rfloor$, and it is pointed from $i = APos[p]$ in A. Thus we find consecutive characters starting at $T_{A[i]}$ until surpassing r. Overall the process takes time $O(r - l + d)$.

4 A Compressed Graph Representation

Our essential idea is to regard the graph $G = (V, E)$ as a *collection of texts* and use a self-index to manage it. If we just used a compressed sequence representation for this collection [SG06, GN06, FMMN06] we would be able of retrieving the neighbors of any node (as this is equivalent to extracting its list of nodes as a substring of the text collection). Yet, by using a compressed self-index, we will be able to retrieve, in addition, the reverse neighbors, by using the search capabilities of the text index.

Definition (*graph text*) Given a labeled directed graph $G = (V, E)$, let $N(v) = \{u \in V,\ (v, u) \in E\}$ the set of neighbors of node $v \in V$. Let $T(v)$ be the sequence formed by the nodes in $N(v)$ in decreasing indegree order. Let $v_1,\ v_2,\ \ldots v_n$ be the set V listed in an decreasing indegree order. Then, the text of graph G is the string $T(G) = T(v_1)T(v_2)\ldots T(v_n)$ over the alphabet V.

Note that the number of occurrences of v_i in $T(G)$ is precisely its indegree, and $|T(G)| = |E| = e$. Our compressed graph representation is essentially a CSA built over $T = T(G)\$$, yet we need some specialized structures. More precisely, our data structures are as follows:

- Ψ: is the usual Ψ function described in the previous section. The only difference is that, if t_i is the last character of $T(v_j)$, then $\Psi(i)$ is defined as $|T| + j$.
- H: is a pointer to the first (header) node of each list in the suffix array A of T (A is not represented explicitly). That is, $H[i] = j$ iff $t_{A[j]}$ is the first character of $T(v_i)$. H has $n = |V|$ entries.
- D: is the usual D array (equipped for *rank* queries) described in the previous section. Note that we do not have array S.

As explained, obtaining the neighbors of a node is equivalent to retrieving a substring of T, whereas obtaining the reverse neighbors is similar to a search query. More precisely, the processes are as follows:

To obtain the neighbors of v_i: We get the entry $j = H[i]$ in A that points to the first character in $T(v_i)$. The first element of $T(v_i)$ is thus $v_{rank(D,j)}$, just as in the previous section. Note that we do not need S because the nodes are sorted by decreasing indegree, that is, by decreasing number of occurrences. Therefore, all those nodes with zero occurrences are at the end, and thus the t-th bit set in D directly corresponds to v_t. Once we obtain the first character of $T(v_i)$, we move on to obtain its next character from $j' = \Psi(j)$, and so on. The process finishes when $j > e + 1$, as this signals the end of the list. This method retrieves each neighbor in constant time, yet it cannot tell the outdegree of v_i (that is, $|T(v_i)|$) without actually generating the whole list. It can, on the other hand, generate only a prefix of the list in time proportional to the length of the prefix generated. The neighbors are actually output from most to least popular (higher to lower indegree).

To obtain the reverse neighbors of v_i: Those are the nodes j in whose lists $T(v_j)$ does v_i appear. Their number corresponds to a counting query for the pattern $P = v_i$. This is a length-1 pattern, which is easily counted as $ep - sp + 1$, where $ep = select(D, i + 1)$ and $sp = select(D, i)$ (because all the occurrences of v_i are contiguous in A). Thus we can compute the indegree of a node in constant time. In order to find out the reverse neighbors, we must locate all those occurrences $sp \le i \le ep$. For each such $A[i]$, more than its position in T (as a usual locating query) we want to know the list $T(v_j)$ to which $A[i]$ belongs. For this sake we apply Ψ consecutively over i (thus virtually advancing forward in $T(v_j)$), until we reach the end of the list. This is recognized because we reach at $i > e + 1$, and moreover $j = e + 1 - i$ (see our modified Ψ definition). The time to output each reverse neighbor is in this case proportional to its outdegree. This can be reduced by using a sampling technique similar to that of the previous section for the basic CSA.

The overall space usage of our index is $n\lceil\log(e+1)\rceil$ bits for H, $e+o(e)$ bits for D, and the space for Ψ. This can easily be made $eH_0(G) + O(e \log\log n)$ by using the same technique of Sadakane (see previous section). By compressing the runs we can achieve space of the form $O(eH_k(T) \log n)$, proportional to the k-th order entropy of $T(G)$. Note that, by regarding the graph as a text, we can give some space bound in terms of the k-th order entropy of that text representation, and moreover, the property of similarity in neighbor lists successfully translates into k-th order text compression. Note also that we are not wasting any space in order to support reverse neighbors.

5 Preliminary Experimental Results

We implemented our compressed graph data structure (see Appendix A for some details) and applied over different graphs obtained from public sources such as WebBase[4] and WebGraph[5]. In this section we compare our structure against the best alternative, that is, the structure of Boldi and Vigna [BV03]. To display comparable results, we indexed the same `.uk` crawl mentioned in their paper ($n = 18.5$M, $e = 292$M).

Our machine is an Intel Pentium 4 at 3GHz with 4GB RAM, running Linux. Our code is written in C and compiled with `gcc` using full optimization.

Figure 1 compares the performance of both schemes. In our case we have varied the sampling step q from 8 to 256, obtaining different space/time tradeoffs. In addition, reverse neighbor counting takes slightly less than 1 microsecond independently of the sampling step. The times for the other scheme are obtained from their paper [BV03]. Those times are not fully comparable because different machines and programming languages are used, but still one can see that the times are of about the same order of magnitude for the forward neighbors, where both schemes behave similary when using 8–10 bits per edge. For reverse neighbors, we are still far behind.

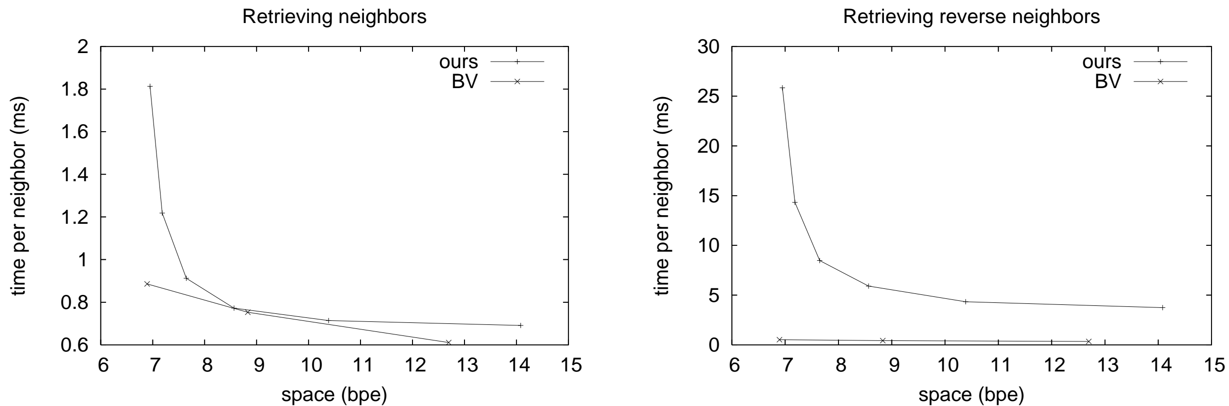

Figure 1: Space/time tradeoffs for normal (left) and reverse (right) neighbor search of our scheme and that of Boldi and Vigna.

6 Conclusions

The main contribution of this paper is to open a new line of attack to the graph compression problem, more precisely to implement a compressed graph data structure that can be manipulated without uncompressing it. Our idea has two key ingredients. The first is that a graph can be regarded as a text, where there exist a large number of theoretical and practical tools for compression and indexing. The second is that, using a self-index, we can retrieve the forward and the reverse neighbors using a single data structure: While retrieving forward neighbors is similar to recovering

[4]`http://www-diglib.stanford.edu/~testbed/doc2/WebBase`.

[5]`http://webgraph.dsi.unimi.it`.

a passage of the compressed text, retrieving reverse neighbors is achieved via searching the text. Thus the concepts of compression and indexing are mixed to obtain a succinct data structures for graphs with extended functionality. While alternative approaches need two separate structures for the graph and its transpose, we have a single indivisible structure that simulates both.

We have shown that our technique, even implemented as a first prototype, is at times competitive with the highly-optimized and carefully engineered implementation of Boldi and Vigna, by far the best existing competitor. This shows that our idea has the potential of becoming a fully competitive choice for this problem. Moreover, we have demonstrated that the connection with text compression makes it possible to define a concept of *graph entropy* that captures some of the key features that permit compression of Web graphs, and which are absent in the much weaker traditional definitions of graph entropy.

As explained, we regard this paper as foundational, not terminal. The ideas we have proposed here admit many other realizations. For example, we can obtain a different text from the graph by listing the edges (u, v) instead of the lists of adjacencies. This makes a longer text which, however, promises to be much faster to answer queries[6]. We can also use a different self-index to handle the text, such as [MN05, Nav04]. We are currently pursuing all those combinations. It is also open to explore which other operations can be efficiently answered with these compressed data structures. Finally, as usual, we will have sooner or later to face the challenges of construction in little space, dynamism, and secondary memory friendliness. Definitely, there is a lot of fun ahead.

References

[ACL00] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proc. STOC'00*, pages 171–180, 2000.

[AM01] M. Adler and M. Mitzenmatcher. Towards compressing Web graphs. In *Proc. DCC'01*, pages 203–212, 2001.

[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. WWW'98*, pages 469–477, 1998.

[BBK03] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. SODA'03*, pages 579–588, 2003.

[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. Also in *Proc. WWW9*.

[Bla06] D. Blandford. *Compact data structures with fast queries*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2006. Also as Tech. Report CMU-CS-05-196.

[BV03] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. Manuscript, 2003.

[6] This idea came up in a conversation with Paolo Ferragina.

[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 *Lecture Notes in Computer Science v. 1443*, pages 118–129, 1998.

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

[CPMF04] D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsos. Fully automatic cross-associations. In *Proc. ACM SIGKDD'04*, 2004.

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

[Eli75] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–20, 1975.

[FM05] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[FMMN06] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 2006. To appear. Also as TR 2004-05, Technische Fakultät, Universität Bielefeld, Germany, December 2004.

[GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Vol. of WEA'05*, pages 27–38, Greece, 2005.

[GN06] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. CPM'06*, LNCS 4009, pages 295–306, 2006.

[GRRR04] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. CPM'04*, LNCS 3109, pages 159–172, 2004.

[GV06] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[HKL99] X. He, M.-Y. Kao, and H.-I. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *Journal of Discrete Mathematics*, 12(3):317–325, 1999.

[HKL00] X. He, M.-Y. Kao, and H.-I. Lu. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM Journal of Computing*, 30:838–846, 2000.

[IR82] A. Itai and M. Rodeh. Representation of graphs. *Acta Informatica*, 17:215–219, 1982.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS'89*, pages 549–554, 1989.

[KRRT99] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large scale knowledge bases from the Web. In *Proc. 25th VLDB Conference*, 1999.

[KW95] K. Keeler and J. Westbook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58:239–252, 1995.

[Lu02] H.-I. Lu. Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits. In *Proc. SODA'02*, pages 223–224, 2002.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[MR97] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. FOCS'97*, pages 118–126, 1997.

[MRRR03] J. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. ICALP'03*, LNCS n. 2719, pages 345–356, 2003.

[Mun96] I. Munro. Tables. In *Proc. FSTTCS'96*, LNCS n. 1180, pages 37–42, 1996.

[Nao90] M. Naor. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics*, 28(303–307), 1990.

[Nav04] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[Pag99] R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. ICALP'99*, pages 595–604, 1999.

[RGM03] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proc. IEEE Conference on Data Engineering*, 2003.

[Ros99] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization*, 5(1):47–61, 1999.

[RRR02] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, 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.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[SG06] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA'06*, pages 1230–1239, 2006.

[SY01] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. DCC'01*, pages 213–222, 2001.

[Tur84] G. Turán. Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.

A Practical Implementation

To implement our scheme, some practical decisions have to be made. We describe the most important ones in this section.

Node ordering: Sorting the nodes by decreasing indegree has several advantages. The most obvious one is that we do not need to store string S, as explained (note that S would be significantly long in our application). Another is that it is likely to produce longer repeated substrings in T, and thus enlarge the runs in Ψ. The rationale is to avoid that infrequent nodes interrupt similar sequences produced by similar sets of frequent nodes in the neighbor lists.

Compression of Ψ. The number of cases where $\Psi(i+1) = \Psi(i) + 1$ amounts to a large part of the array (close to 75% in our tests). Thus it is good idea to give some special treatment to the runs (we also verified this experimentally). What we do is that we represent consecutive differences (gaps) $\Psi(i+1) - \Psi(i)$, and when this difference is 1 we assume the start of a run of 1's, so the number that follows is the run length (even if this length is 1). To encode the differences we tried δ-encoding and some relatives but the results were not very good. We opted for an optimal scheme such as Huffman encoding (even if this implies storing its table). By using canonical Huffman encoding we have to store little more than the permutation of values in increasing frequency order.

A problem is that the distribution of values has a very heavy tail, that is, there is a very large number of different gap values that occur only once. Thus we truncated the Huffman coding at some number of symbols. An escape code (whose frequency adds up all the non-represented symbols) indicates that a non-represented symbol follows. This symbol is encoded next to the escape code, in absolute form (rather than difference form) using a fixed number of bits. We use 512K symbols for this Huffman coding and still the escape symbol was always the most frequent one, followed by the gap 1. Run lengths are encoded using another Huffman encoder, where all the possible lengths are represented in the code.

To permit fast access to Ψ we use a technique similar to the practical implementation of Sadakane [Sad03]. At regular intervals q in Ψ we insert absolute samples in its compressed representation, using a fixed number of bits. An array Pos indexed by sample number stores a pointer to the bit position in the compressed representation of Ψ where the corresponding absolute sample is written. In order to obtain $\Psi(i)$ we find $p = \lfloor i/q \rfloor$ and start decompressing Ψ from position $Pos[p]$, skipping entries until reaching position i from pq. Note that a run lets us advance many positions in one shot. In this traversal we accumulate the differences over the original absolute sample, unless we find other absolute values (next to escape codes in the compressed representation). Note also that, if an absolute sample coincides with an escape code, we represent the absolute value only once, without the escape code. The most relevant space/time tradeoff given by the index representation is given by parameter q.

Reducing H. Header array H poses a considerable space overhead, of about 2–3 bits per edge (as each pointer requires about 30 bits in sufficiently large graphs, and the ratio e/n is about 10–15). This can be reduced by considering that in many graphs only a few nodes have positive outdegree (nonempty $T(v_i)$ list). For this sake, we set up a bit vector $isH[1, n]$, where $isH[i] = 1$ iff $|T(v_i)| > 0$, and store H only for the nodes with positive outdegree: if $isH[i] = 1$, the original value $H[i]$ is stored at $H[rank(isH, i)]$. For this to be a bad idea it is necessary that about 29/30 of the nodes have outgoing edges, which is hardly the case in Web graphs.

Rank and select. We need to do *rank* over isH and D, and *select* over D. We use the optimized implementation of [GGMN05], which poses 37% space overhead over the bit array. It answers *rank* in true constant time, whereas *select* is solved in logarithmic time by a layered binary search over the *rank* data structures.

Locate by groups. Locating the occurrences of $[sp, ep]$ one by one is a waste of work, as already noted by Sadakane [Sad03]. We reimplement his idea for our case. Note that, when we locate occurrence $sp + 1$, we have to access $\Psi(sp + 1)$, which is next to $\Psi(sp)$. We most probably have to start from the same sampled position and decompress again the same bits, to reach one code further. It would be much faster to decompress $\Psi(sp, ep)$ in one single pass. Moreover, as runs in Ψ are common, it is usual that positions $\Psi(sp) \ldots \Psi(ep)$ contain a few long runs of consecutive values, so that we can apply again the same idea over a few runs instead of splitting the process into many individual tracings. We avoid the recursion by starting with an array containing $sp \ldots ep$, then detecting its runs (only one run initially), and then applying Ψ over each run separately (one pass of sampling plus decompression per run). The process is now repeated (finding the new runs again, etc.) until all the values exceed $e + 1$. This technique yielded a 10-fold improvement in the time cost of reverse neighbor queries. Actually, with the performance achieved, adding samples to shorten the paths looks as a bad idea, as now many paths are traversed for free, and we would need a rather dense (and thus space-expensive) sampling to achieve a noticeable improvement. Still, reverse queries are 10 times slower than normal neighbor queries.