Compression of Web and Social Graphs supporting Neighbor and Community Queries

Cecilia Hernández
Dept. of Computer Science Univ. of Concepción
Dept. of Computer Science Univ. of Chile
chernand@dcc.uchile.cl

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

ABSTRACT

Motivated by the needs of mining and advanced analysis of large Web graphs and social networks, we study graph patterns that simultaneously provide compression and query opportunities, so that the compressed representation provides efficient support for search and mining queries. We first analyze patterns used for Web graph compression while supporting neighbor queries. Our results show that composing edge-reducing patterns with other methods achieves new space/time tradeoffs, in particular breaking the smallest known space barrier for Web graphs when supporting neighbor queries. Second, we propose a novel graph compression method based on representing communities with compact data structures. These offer competitive support for neighbor queries, but excel especially at answering community queries. As far as we know, ours is the first graph compression method supporting such a wide range of community queries.

Categories and Subject Descriptors

H.2.8 [**Data Representation**]: Data Mining

General Terms

Algorithms, Experimentation, Theory

Keywords

Compression, Web Graphs, Social Networks, Compact Data Structures

1. INTRODUCTION

Much information on the structure, meaning and usage of the Web can be extracted by analyzing its graph. Web graphs are crucial for ranking algorithms, such as PageRank [7] and HITS [22], as well as for spam detection [3]. On the other hand, as never before, much information on social behavior is digitally available thanks to technologically supported social networks such as Facebook, Flickr, and many others. Current research on social networks includes detecting relevant communities, discovering important actors and understanding how the information flows in the network [21, 12, 27]. Web graphs and social networks are expanding in size. In 2008, Google reported more than 1 trillion unique URLs[1] on the Web, and Facebook reached 500 million active users in July 2010[2]. The scale of these networks has brought new challenges for analyzing and mining large graphs.

The research community has proposed compressed structures with direct access capabilities to facilitate mining and analysis of these large graphs. Compression rate is typically expressed as total number of bits per edge (bpe). The WebGraph framework [5, 4] is usually used as a reference on Web graph compression. Frequently, queries are reduced to the most basic one of listing the out-neighbors of a node, but in some cases in-neighbors are considered as well. Only recently, compression of social networks has been proposed, with out-neighbor [13] and with out/in-neighbor query support [26]. Even though out/in-neighbor queries may be used to build more complex operations such as community and outlier detection, we are not aware of any work studying the implementation and space/time requirements of such complex operations.

In this paper, we first analyze different graph patterns used for compressing Web graphs while supporting neighbor queries. We are particularly interested in analyzing whether the use of edge-reducing patterns [15, 11] might be combined with node ordering methods [5, 4, 2], and still take advantage of the similarity and locality found in graphs [5, 4, 2]. This combination has not been considered before. Our results show that it is possible to combine these methods to improve upon the best known results in space/time efficiency for compressing and supporting queries on Web graphs. In particular, we improve significantly upon the smallest spaces reported in the literature for Web graphs.

Our second contribution is a novel graph compression method based on finding communities and representing them with compact data structures. The resulting representation answers out/in-neighbor queries. However, the most interesting aspect is that it is very efficient at answering various community queries, which are useful for mining Web and social network graphs. The communities we consider are bicliques, which model node-centric communities based on complete mutuality [1] and have shown to be meaningful for mining [11]. Bicliques have already been used for Web graph

The 5th SNA-KDD Workshop '11 (SNA-KDD'11) August 21, 2011, San Diego CA USA.

[1] `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`
[2] `http://blog.facebook.com/blog.php?post=409753352130`

compression [11], but we introduce a different representation that permits querying those communities efficiently. Some of the queries we consider are: "enumerate the communities where entity X participates", "get the members that belong to a community Y", and "enumerate the communities with their sizes and densities". Querying over communities might be of great interest for discovering relevant actors based on positions and community sizes and densities where actors participate. Several measures consider prestige and centrality based on position and node degree [21, 27]. Unlike other compressed structures, our representation indexes internal graph patterns found in graphs (bicliques), enabling a wide set of query operations. We also index the rest of the graph (without bicliques), which facilitates its access and analysis.

2. RELATED WORK

The Web is one of the most studied graphs, and compressing it has been an active research area [10, 32, 5, 15, 11, 4]. Randall et al. [29] first proposed lexicographic ordering of URLs to exploit locality and similarity for compressing Web graphs. Later, Boldi and Vigna [5] proposed the WebGraph framework, one of the most competitive approaches in terms of space/time requirements for out-neighbor queries. This approach exploits power-law distributions, similarity and locality. They also developed ς codes, which are well suited for compressing power-law distributed data with small exponents [6]. More recently Boldi et al. [4] (BV) explored and evaluated other ordering methods, including Gray ordering, to improve their previous results.

Another recent compression scheme, by Apostolico and Drovandi [2], reorders the nodes based on a breadth-first (BFS) traversal of the graph, instead of the lexicographic order. It then encodes the outdegrees of the nodes in the order given by the BFS traversal, plus a list of the edges that cannot be deduced from the BFS tree. It achieves compression by dividing those lists into chunks and taking advantage of locality and similarity. We refer to this approach as AD.

Buehrer and Chellapilla [11] used a scalable pattern mining approach to provide compression of Web graphs. They used min-wise independent hashing [9] for clustering and identified directed complete bipartite graphs (i.e., bicliques) using a frequent itemset mining approach on each cluster. A biclique is formed by two sets of nodes, A and B, such that all the elements of A point to all the elements of B. For each biclique, they defined a virtual node that connects the two sets, replacing all the $|A| \cdot |B|$ links from A to B by $|A|$ links from A to the virtual node, plus $|B|$ links from the virtual node to B. Applying gap encoding to the resulting graph, they were able to improve the original compression of Boldi and Vigna [5]. We refer to this scheme as VNM (Virtual Node Miner). Recently, VNM has been used to improve running times for Web graph algorithms based on random walks, such as PageRank and HITS, achieving speedups proportional to the compression ratio [20]. Another edge-reducing approach was proposed by Claude and Navarro [15]. It is based on Re-Pair [25], a grammar-based compressor. Re-Pair repeatedly finds the most frequent pair of nodes in the concatenated sequence of all adjacency lists and replaces it with a new symbol.

All these methods provide efficient out-neighbor navigation, that is, retrieving the adjacency list of any node. Adding in-neighbor navigation (i.e., retrieving the nodes that point to a node) is usually performed by representing the transpose of the graph in addition to the graph itself. Boldi et al. [4] showed that Gray ordering outperforms others for the transpose of Web graphs. A proposal by Brisaboa et al. [8] supports out/in-neighbor navigation using a structure that represents the adjacency matrix taking advantage of its sparseness. A recent implementation improvement is available [24]; we refer to it as k2tree. Claude and Navarro [16] presented a compact Web graph representation that enrichs the output of Re-Pair compression with compact data structures (Re-Pair-GMR) [17] that yields in-neighbor queries as well.

Recent works on compressing social networks [13, 26] have exposed compression opportunities, although in less degree than in Web graphs. The approach by Chierichetti et al. [13] is based on the Webgraph framework [5], using shingling ordering (based on Jaccard coefficient) [9] and exploiting link reciprocity. Maserrat and Pei [26] achieve compression by defining a Eulerian data structure using multi-position linearization of directed graphs. Their approach supports out/in-neighbor queries in sublinear time.

In the context of communities, Saito et al. [30] presented a method for spam detection based on classical graph algorithms such as identification of weak and strong connected components, maximal clique enumeration and minimum cuts. Other techniques, based on graph algorithms, aim to extract small subgraphs or small communities for mining purposes [23]. The Web graph compression proposed by Buehrer and Chellapilla [11] use the notion of communities defined by bicliques. They provide community seed semantic evaluation showing four community patterns found during compression. However, none of these schemes supports community queries on the compressed structure.

3. COMPRESSING WEB GRAPHS

In this section we describe edge-reducing patterns (VNM [11] and Re-Pair [15]), node ordering methods (BV [6, 4] and AD [2]), and techniques that support out/in-neighbor queries such as k2tree [8]. We present experimental results on the effect of combining edge-reducing patterns with the other methods and discuss our results.

3.1 Edge-reducing compressors

The VNM [11] compressor is based on the idea of identifying bicliques and using virtual nodes as one level of indirection between the two sets. The use of virtual nodes reduces the number of edges. After reducing edges the authors apply gap and Huffman coding for compressing Web graphs. Identifying bicliques has two phases: clustering and mining. The clustering phase groups similar rows of the adjacency matrix. It uses the heuristic of grouping rows with high Jaccard coefficients [9]. For a graph $G = (V, E)$, they use k min-wise independent hash functions [9] to obtain a hash function matrix of size $k \cdot |V|$. Rows in the matrix are sorted lexicographically and then traversed by column, grouping rows with the same value. When the number of rows drops below a threshold, a new cluster is defined. The hash function matrix is only used for clustering.

The mining phase operates locally within each cluster, looking for vertices with common subsets of outlinks. More relevant communities are given by larger outlink sets (of size *pattern_size*) shared by larger vertex sets (of size *pattern_frequency*), which provide better compression. The

mining algorithm builds a trie on the adjacency lists of the vertices, and chooses long paths in the trie to find good sets.

The algorithm performs successive iterations, so that virtual nodes created during an iteration can participate in bicliques in a subsequent iteration.

Another edge-reducing compressor is based on Re-Pair [25], a grammar-compression algorithm consisting of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol. Starting from an integer sequence S, the algorithm iterates over the following steps. (1) It identifies the most frequent pair ab in S. (2) It adds the rule $r \to ab$ to a dictionary R, where r is a new symbol that is not in S. (3) It replaces any occurrence of ab by r in S. This is repeated until the replacements do not improve compression. The output of the algorithm is the remaining sequence C and the dictionary R. C is formed with both terminal symbols (original symbols in S) and nonterminal symbols (introduced in step 2). In order to obtain the original symbols, nonterminals must be decompressed using the information stored in dictionary R.

Re-Pair has been used for compressing the Web graph adjacency lists, providing competitive results in terms of compression, and supporting fast out-neighbor queries [15].

3.2 Reordering nodes

The WebGraph framework [4] supports different Web page orderings (URL, lexicographic, Gray ordering, loose and strict host-by-host Gray ordering). With either ordering, each Web page is mapped to a unique integer identifier. WebGraph then exploits locality and similarity. Locality means that, usually, most of the links of a page point to nearby pages. Similarity means that many Web pages tend to have many outlinks in common, or in other words, that some adjacency lists are very similar to others.

WebGraph uses two compression parameters, *window_size* (w) and *maximum_reference_count* (m). The *window_size* corresponds to the number of previous rows in the adjacency matrix considered when compressing a row x for reference coding. The idea is to find the most similar previous row in that window. If that row exists, it is called a prototype. A larger *window_size* yields better and slower compression. On the other hand, the *maximum_reference_count* is the maximum allowed length of a reference chain. When compressing a row x of the adjacency matrix, the compression is done with respect to a previous prototype y, and row y could have been compressed differentially with respect to some other prototype z, generating a reference chain. Limiting the length of the reference chain retains fast decompression.

Each row is encoded using its prototype, if any, plus outlinks not in the prototype. WebGraph encodes consecutive outlinks using interval encoding and ς codes for integers [6].

Apostolico and Drovandi [2] proposed an ordering method based on breadth-first search. The method depends on the topological structure of graphs instead of the URLs. They use gap and run-length encoding and define a new integer encoding called π codes, claimed to be better suited than ς codes for power-law distributions with exponent close to 1.

The compression scheme works on chunks of *level* (l) nodes. Parameter *level* provides a tradeoff between compression performance and time to retrieve the adjacency list of a node. With $l = 8$ they achieve better compression and similar access times than WebGraph.

3.3 Compressors supporting out/in-neighbors

These compressors support out/in-neighbor queries on the same structure, avoiding the need of using the graph plus its transpose (as it is the case with the WebGraph, VNM, and Re-Pair compressors).

The k2tree scheme [8] represents the adjacency matrix by a k^2-ary tree of height $h = \lceil \log_k n \rceil$ (where n is the number of vertices). Simulating an MX-Quadtree decomposition [31], it divides the matrix into k^2 submatrices of size n^2/k^2. The tree representation, at each level, defines k^2 child nodes containing a bit "1" if there is at least a "1" on the submatrix it represents, and a "0" otherwise. Nodes represented with a bit "1" are recursively divided into k^2 nodes. At the last level, the leaf nodes contain the bits of the adjacency matrix. A recent improvement [24] uses statistical compression at the last level and retains fast access times.

Re-Pair GMR [16] uses the grammar-based compression technique Re-Pair, and defines compact data structures based on bitmaps and sequences with fast rank/select operations [14, 17]. These support in-neighbor queries by finding in the adjacency lists all the positions where the node, or a nonterminal expanding to it, is mentioned.

3.4 Composing methods

In this section we evaluate the impact of combining edge-reduction with other methods. We proceed in two stages: an edge-reduction stage yields a new graph, containing fewer edges and more nodes (including virtual nodes). Then a compression stage applies existing compression techniques on the edge-reduced graph.

For the edge-reducing stage we consider three alternatives. The first is VNM, which we implemented as described in its article [11], using C++ and STL, but we did not implement the compression of the resulting graph (the full method, including this last compression, will be called VNM_b). We added a compression parameter $maximum\_saving = pattern\_size \cdot pattern\_frequency$. This parameter defines the minimum biclique size we consider for edge reduction. Second, we implemented Re-Pair as just an edge-reducing method (i.e., rule $r \to ab$ is seen as the creation of a virtual node r with edges to a and b), which we call RP_o. Third, we consider VNMRP, which applies VNM and then RP_o.

For the compression stage we also considered three alternatives. First, we used the full Re-Pair compression scheme obtained from its authors [15] (RP_c). Second, we used WebGraph (BV), version 2.4.4 taken from `http://webgraph.dsi.unimi.it`. Third, we used AD version 0.2.1, without dependencies, taken from `http://www.dia.uniroma3.it/~drovandi/software.php`.

We executed our experiments on a Linux PC with 8 processors Intel Xeon at 2.4GHz, with 32 GB of RAM and 12 MB of cache, using sequential algorithms.

We used the following data sets, corresponding to real Web graphs and social networks; some statistics are given in Table 1. Web graphs are snapshots made available at `http://law.dsi.unimi.it` within the WebGraph project. As social network graphs, we use Facebook (undirected graph, from `http://socialnetworks.mpi-sws.org/data-wosn2009.html`) and Flickr (directed graph, from `http://socialnetworks.mpi-sws.org/data-www2009.html`) data sets used in social network research [33, 27]. The LiveJournal (directed graph) data set from the SNAP project (Stanford Network Analysis Package,

Data Set	Nodes	Edges
eu-2005	862,664	19,235,140
in-2004	7,414,866	194,109,311
uk-2002	18,520,486	298,113,762
ar-2005	22,744,080	639,999,458
it-2004	41,291,594	1,150,725,436
Facebook	45,622	817,090
Flickr	1,861,232	22,613,981
LiveJournal	4,847,571	68,993,773

Table 1: Some statistics of our test graphs. Here in-2004 corresponds to indochina-2004 and ar-2005 to arabic-2005.

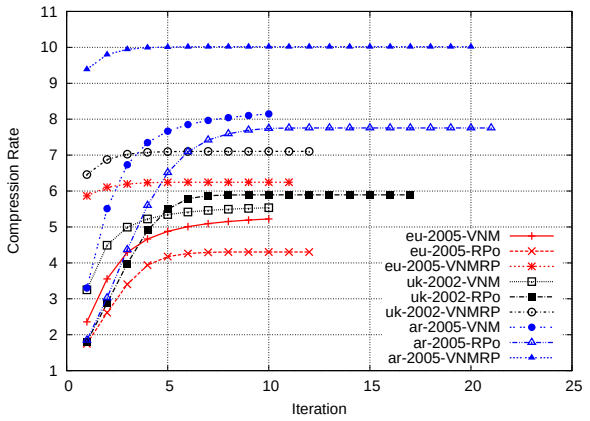

Figure 1: Compression rate based on number of edges.

`http://snap.stanford.edu/data/`).

The first experiment evaluates the compression rate defined as the total number of edges of the original graph divided by the edges remaining after compression (including from/to virtual nodes). Figure 1 shows the compression rate achieved by VNM, RP_o, and by VNMRP. The results suggest that combining both techniques provides better compression rates than applying either of them individually.

The second experiment measures the compression in terms of *bpe* and aiming at maximum compression (i.e., no structures for direct access to the graphs are maintained). We found that reducing the number of edges in the edge-reducing stage does not always improve the final compression figures, as shown in Table 2. For this table we combined edge-reduction using VNM with compression using BV and AD. We tuned VNM using parameter m. VNM* ($m = 1$) allows any virtual node pattern that reduces edges, thus minimizing the edges as much as possible in the edge-reducing stage. Table 2 shows that VNM* does not achieve the lowest possible bpe values, but these are achieved with $m = 30$ on `eu-2005`, and $m = 100$ on the other Web graphs (those are used in the row labeled VNM). The best compression is achieved with loose host-by-host Gray ordering of BV with $w = 8$ and $m = -1$ (for maximum compression). This suggests that there are sufficient regularities that BV can exploit after removing redundant edges, even if there are more nodes in the graph.

We also apply AD with $l = 100$ after VNM. Larger l did not yield more compression. In this case, it is best to run VNM reducing the edges as much as possible (VNM*). Once again, the combination works better than the individual techniques. In particular, VNM+BV and VNM*+AD *beat the best known space for compressing Web graphs.*

Name	eu-2005	in-2004	uk-2002	ar-2005	it-2004
BV	3.60	1.14	1.86	1.50	1.61
VNM^*+BV	2.61	1.29	2.01	1.64	1.52
VNM+BV	**2.24**	**1.04**	**1.65**	**1.35**	**1.28**
AD	2.89	1.08	1.90	1.67	1.55
VNM^*+AD	**2.13**	**1.00**	**1.68**	**1.36**	**1.32**

Table 2: Maximum compression comparison in bpe for combining VNM with BV and AD.

Name	eu-2005	in-2004	uk-2002	ar-2005	it-2004
VNM_b	2.90	-	1.95	1.81	1.67
RP_c	4.98	2.62	4.28	3.22	2.91
VNM*+RP_c	3.57	2.30	3.65	2.78	2.63
BV	4.49	1.88	2.82	2.26	2.37
VNM+BV	2.87	1.51	2.42	1.87	1.76
AD_8	3.68	1.61	2.64	2.28	2.17
RP_o+AD_8	3.15	1.71	2.67	2.01	2.26
VNMRP+AD_8	2.28	1.17	1.92	1.50	1.49
VNM*+AD_4	2.39	1.24	2.05	1.57	1.57
VNM*+AD_8	**2.26**	**1.12**	**1.87**	**1.47**	**1.45**

Table 3: Compression in bpe when allowing random access, for various methods. VNM_b are the bpe values reported by Buehrer and Chellapilla [11].

Table 3 shows the compression results when the graphs contain structures to support direct access. We include bare compression methods (VNM_b, RP_c, BV, and AD) and our best performing combinations. To enable direct access we use BV with $w = 8$ and $m = 3$, and AD with $l = 4$ and 8. Since both BV and AD exploit locality and similarity, the results suggest that the BFS ordering used in AD works better than the loose host-by-host Gray ordering used in BV. We achieve the best compression results using VNM*+AD_8.

Tables 4 and 5 give average times to retrieve adjacency lists of 20 million random nodes. In the first we do not include the time to recursively expand the virtual nodes and the node id mapping to obtain the original graph. Average times displayed in Table 4 are important considering recent research where using VNM allows speeding up Web graph algorithms based on random walk such as PageRank [20]. In Table 5 we include all the required operations to retrieve the original adjacency lists. We can see that adding the VNM* preprocessing using AD multiplies access times by 2–3, yet these are still within 20 microseconds per node, that is, below the microsecond per delivered edge. Using VNM with BV multiplies access times only by 2.

3.5 Bidirectional navigation

We additionally consider combining edge-reduction methods with techniques that support out/in-neighbor queries. We evaluate the following variants:

T1: Re-Pair GMR [16] on the original graph.
T2: k2tree [8] on the original graph.

Data set	AD		VNM*+AD		BV	VNM+BV
	$l = 4$	$l = 8$	$l = 4$	$l = 8$	$m = 3, w = 8$	
eu-2005	2.50	3.86	0.78	1.26	1.50	1.03
in-2004	2.02	2.92	0.75	1.12	1.41	1.12
uk-2002	1.75	2.61	0.83	1.14	1.34	1.19
ar-2005	2.54	3.65	0.80	1.22	1.58	1.34
it-2004	2.41	3.66	0.79	1.30	1.78	1.35

Table 4: Average time to retrieve an adjacency list, in microseconds, without any reordering nor expansion.

Data set	AD		VNM*+AD		BV	VNM+BV
	$l = 4$	$l = 8$	$l = 4$	$l = 8$	$m = 3,\ w = 8$	
eu-2005	4.36	6.63	10.9	19.93	2.18	4.71
in-2004	3.58	5.23	5.65	10.72	1.80	3.33
uk-2002	2.50	3.77	4.53	7.72	1.64	2.90
ar-2005	3.82	5.93	9.49	16.95	2.07	4.53
it-2004	3.70	5.93	7.87	14.23	2.14	4.18

Table 5: Average time to retrieve an adjacency list, in microseconds, considering full expansion and reordering.

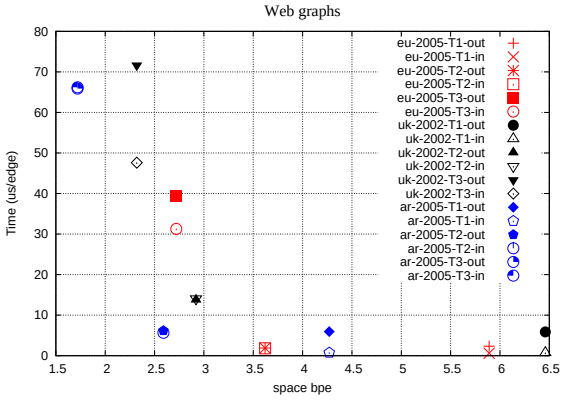

Figure 2: Space/time efficiency with out/in-neighbor queries with T1,T2, and T3 on Web graphs.

T3: VNM on the original graph and then k2tree.

The space/time requirements of these techniques, on Web graphs and social networks, are displayed in Figures 2 and 3, respectively. The combination of VNM and k2tree (T3) achieves by far *the best space efficiency on Web graphs when supporting bidirectional neighbors*. However, this comes at a significant price in access time. The combination, on the other hand, does not work well on social networks, where k2tree alone (T2) is unbeaten. This suggests that reducing the number of edges on social networks removes some degree of sparseness from the original adjacency matrices, which does not happen on Web graphs.

3.6 Discussion

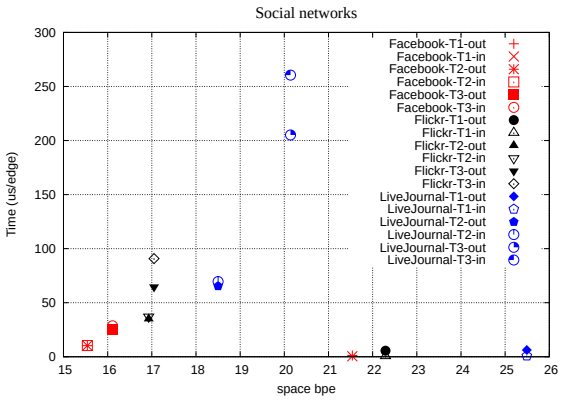

Figure 3: Space/time efficiency with out/in-neighbor queries with T1,T2, and T3 on social networks.

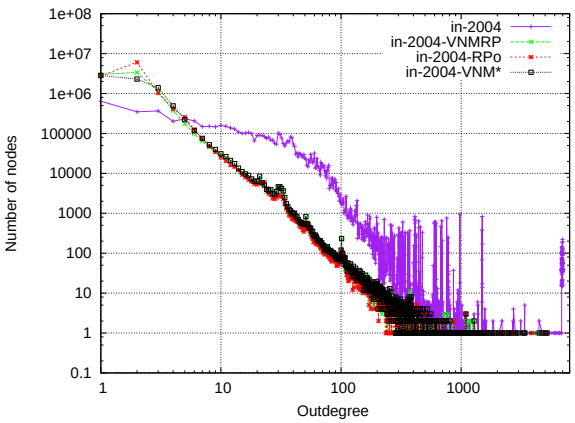

Figure 4: Node degree distribution of in-2004

Data set	Name	edges	bpe
eu-2005	RP_o	0.72	0.71
	VNMRP	0.98	0.99
in-2004	RP_o	0.79	0.65
	VNMRP	0.98	0.95
uk-2002	RP_o	0.74	0.70
	VNMRP	0.98	0.97
ar-2005	RP_o	0.75	0.73
	VNMRP	0.98	0.98
it-2004	RP_o	0.80	0.64
	VNMRP	0.98	0.97

Table 6: Ratios of edge reduction achieved by various methods versus VNM, and of bpe reduction using AD after them.

Our results show that edge reduction, combined with other methods, improves the state-of-the-art compression of Web graphs. Figure 4 shows the node degree distribution of the in-2004 original graph and after applying VNM, RP_o, and VNMRP. Very similar figures are obtained for the other graphs. We observe that reducing edges produces very clean power law distributions in the node degrees, regardless of the edge-reduction method used. However, this new shape does not seem to have, in general, an impact on the compression achieved with AD, beyond the mere reduction in the data size. Table 6 shows the reduction in number of nodes plus edges achieved by the edge-reduction methods as a fraction of VNM, and the corresponding reduction in bpe after applying AD. It can be seen that in most cases the reduction factors are very similar, that is, AD achieves the same reduction in space on the original and on the reduced graphs, and thus the overall space improvement over bare AD is a consequence of working on a smaller graph and not of the different node degree distribution.

Exceptions to this rule are in-2004 and it-2004, where AD obtains a significant further reduction after applying RP_o. This may be due to the fact that RP_o creates many virtual nodes of outdegree 2, which is possibly advantageous for a BFS ordering. In any case, this shows that further research is necessary to understand the interactions between edge-reduction and edge-reordering methods.

4. COMPRESSING COMMUNITIES

In this section we present a novel compressed data structure based on mining communities. We first provide some

basic notions of compact data structures we will need.

4.1 Compact data structures

A compact data structure provides the same abstraction as its classical counterpart, using little space and supporting interesting queries without having to expand the whole structure. Many compact data structures use as a basic tool a bitmap supporting rank/select/access query primitives. Operation $rank_B(b, i)$ on the bitmap $B[1, n]$ counts the number of times bit b appears in the prefix $B[1, i]$. Operation $select_B(b, i)$ returns the position of the i-th occurrence of bit b in B (and $n + 1$ if there are no i b's in B). Finally, operation $access_B(i)$ retrieves the value $B[i]$. A solution requiring $n + o(n)$ bits and providing constant time for rank/select/access queries was proposed by Clark [14] and a good implementation is available (RG) [18]. In later work, Rao et al. (RRR) [28] improved the required space to $nH_0(B) + o(n)$ bits. $H_0(B)$ corresponds to the zero-order entropy of bitmap B, $H_0(B) = \frac{n_0}{n} \log \frac{n}{n_0} + \frac{n_1}{n} \log \frac{n}{n_1}$, where B has n_0 zeros and n_1 ones.

The bitmap representations can be extended to compact data structures for sequences $S[1, n]$ over an alphabet Σ of size σ. The wavelet tree (WT) [19] supports rank/select/access queries in $O(\log \sigma)$ time. It uses bitmaps internally, and its total space is $n \log \sigma + o(n) \log \sigma$ bits if representing those bitmaps using RG, or $nH_0(S) + o(n) \log \sigma$ bits if using RRR, where $H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$, being n_c the number of occurrences of c in S. Another sequence representation (GMR) [17] uses $n \log \sigma + n\, o(\log \sigma)$ bits, and supports *rank* and *access* in time $O(\log \log \sigma)$, and *select* in constant time.

4.2 Community-based compressed structure

We focus on detecting communities on Web graphs and social networks so that we can represent the original graph in terms of a set of *community graphs* and a *remaining graph*.

Definition 1. *Community.* We define a community as a complete (directed or undirected) bipartite graph, $H(S, C) = G(V = S \cup C, E = S \times C)$. Vertices $s \in S$ are called *sources* and vertices $c \in C$ are called *centers*. We define the community *size* as $|S| + |C|$.

Definition 2. *Community Density.* We define the density by considering the connections inside a community group [1]: $H(S, C) = G(V, E)$ is γ-*dense* if $\frac{|E|}{|V|(|V|-1)/2} \geq \gamma$.

Definition 3. *Directed (Undirected) Bipartite Partition, DBP (UBP).* Let $G(V, E)$ be directed (undirected). A bipartite partition of G consists of a class $\mathcal{H} = \bigcup H_r$ of bipartite graphs $H_r = H(S_r, C_r)$, and a *remaining* graph $\mathcal{R}(V_R, E_R)$, so that all H_r and $\mathcal{R}$ are edge-disjoint and $G = \mathcal{H} \cup \mathcal{R}$.

Definition 4. *Undirected plus Directed Bipartite Partition, UDBP.* Let $G(V, E)$ be a directed graph. We derive from G an undirected graph $G_u(V_u, E_u)$, containing an undirected edge per pair of reciprocal edges in G. Now consider the *UBP* of G_u into $\mathcal{H}_u$ and $\mathcal{R}_u(V_{R_u}, E_{R_u})$. Define $dup(E)$ as the set of directed edges formed by a pair of reciprocal edges per undirected edge in E. Then we call $G_d(V_d, E - dup(E_u - E_{R_u}))$ the remaining directed graph of G. Now consider the *DBP* of G_d into $\mathcal{H}_d$ and R_d. The *UDBP* of G is formed by $\mathcal{H}_u$, $\mathcal{H}_d$, and $\mathcal{R}_d$.

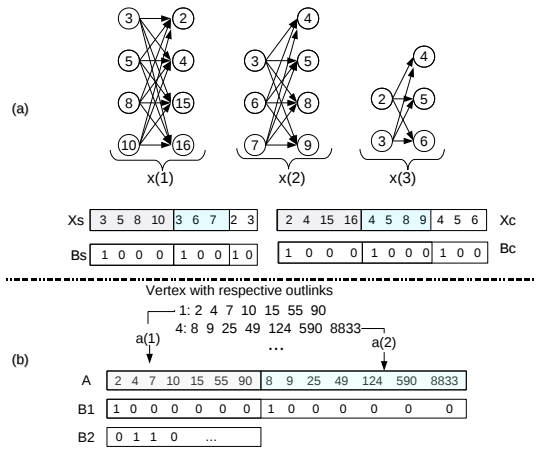

Figure 5: Compact representation of $\mathcal{H}$ (a) and $\mathcal{R}$ (b).

Note that *DBP* and *UBP* aim at representing a graph as a set of communities, regarded as directed or undirected bicliques. A large community $H(S, C)$ will allow us replacing $|S| \cdot |C|$ edges by just $|S| + |C|$ edges and a new node. *UDBP* is more sophisticated, and aims at exploiting *reciprocity* in directed graphs, that is, reciprocal edges. It first looks for reciprocal bicliques, and only then for directed bicliques. Whether *UDBP* is better or worse than plain *DBP* on a directed graph will depend on its degree of reciprocity.

Our construction proceeds in two phases. The first phase is community detection and extraction, and the second is building a compressed structure representation. We use the VNM scheme [11] for detecting and extracting communities. We apply VNM iteratively, extracting large communities first, using higher to lower *maximum_saving* values.

4.3 Compact representation of $\mathcal{H}$

Let $\mathcal{H} = \{H_1, \ldots, H_N\}$ be the communities found in either of the previous definitions. We represent $\mathcal{H}$ as two sequences of integers with corresponding bitmaps. Sequence X_s with bitmap B_s represent the sequence of sources of the communities and sequence X_c with bitmap B_c represent the respective centers. More precisely, we have $X_s = x_s(1)x_s(2)...x_s(r)...x_s(N)$, where $x_s(r) = s_1...s_k$ represents the set S_r of $H_r = (S_r, C_r)$, $s_i \in S_r$, $s_i < s_{i+1}$ for $1 \leq i < k$, and $B_s = 10^{|S_1|-1}...10^{|S_N|-1}1$. In a similar way, we have $X_c = x_c(1)x_c(2)...x_c(r)...x_c(N)$, where $x_c(r) = c_1...c_m$ represents the set C_r, $c_j \in C_r$, $c_j < c_{j+1}$ for $1 \leq i < m$, and $B_c = 10^{|C_1|-1}...10^{|C_N|-1}1$. Figure 5 (a) shows an example.

We represent integer sequences and bitmaps with compact data structures that support rank/select/access operations: we use WTs [19] for sequences and an uncompressed representation [14] for bitmaps, for a total space of $|X|(H_0(X) + 1) + o(|X| \log \sigma)$, where σ is the number of vertices in $\mathcal{H}$. Note that $|X|$ is the sum of the sizes of the communities in $\mathcal{H}$, which can be much less than the number of edges in the subgraph it represents.

We answer out/in-neighbor queries as follows. Their complexity is $O((|output| + 1) \log \sigma)$, which is essentially optimal up to a factor of $O(\log \sigma)$, where σ is the number of nodes in the graph.

out-neighbors (u)
$occur \leftarrow rank_{X_s}(u, |X_s|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_s}(u, i)$
  $p \leftarrow rank_{B_s}(1, y)$
  $s \leftarrow select_{B_c}(1, p)$
  $e \leftarrow select_{B_c}(1, p+1) - 1$
  for $j \leftarrow s$ **to** e **do**
    $out.add(access_{X_c}(j))$
  end for
end for

in-neighbors (u)
$occur \leftarrow rank_{X_c}(u, |X_c|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_c}(u, i)$
  $p \leftarrow rank_{B_c}(1, y)$
  $s \leftarrow select_{B_s}(1, p)$
  $e \leftarrow select_{B_s}(1, p+1) - 1$
  for $j \leftarrow s$ **to** e **do**
    $in.add(access_{X_s}(j))$
  end for
end for

We answer community queries in $\mathcal{H}$ as follows. Table 7 gives the time complexities achieved. Most of them are, again, optimal up to factor $O(\log \sigma)$. The exception is Q7, which can be costlier due to repeated results.

Q1 Get the *centers* of community x.

$start \leftarrow select_{B_c}(1, x)$
$end \leftarrow select_{B_c}(1, x+1) - 1$
for $i \leftarrow start$ **to** end **do**
  $centers.add(access_{X_c}(i))$
end for

Q2 Get the *sources* of community x.

$start \leftarrow select_{B_s}(1, x)$
$end \leftarrow select_{B_s}(1, x+1) - 1$
for $i \leftarrow start$ **to** end **do**
  $sources.add(access_{X_s}(i))$
end for

Q3 Get communities where u participates as a source.

$occur \leftarrow rank_{X_s}(u, |X_s|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_s}(u, i)$
  $p \leftarrow rank_{B_s}(1, y)$
  $comms.add(p)$
end for

Q4 Get communities where u participates as a center.

$occur \leftarrow rank_{X_c}(u, |X_c|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_c}(u, i)$
  $p \leftarrow rank_{B_c}(1, y)$
  $comms.add(p)$
end for

Q5 Get the number of communities where u participates as a source and as a center.

$ncs \leftarrow rank_{X_s}(u, |X_s|)$
$ncc \leftarrow rank_{X_c}(u, |X_c|)$

Q6 Enumerate the members of community x.

$ss \leftarrow select_{B_s}(1, x)$
$es \leftarrow select_{B_s}(1, x+1) - 1$
for $i \leftarrow ss$ **to** es **do**
  $members.add(access_{X_s}(i))$
end for
$sc \leftarrow select_{B_c}(1, x)$
$ec \leftarrow select_{B_c}(1, x+1) - 1$
for $i \leftarrow sc$ **to** ec **do**
  $members.add(access_{X_c}(i))$
end for

Q7 Enumerate the out-communities at distance 1 of community x.

$centers \leftarrow Q1(x)$
for c in $centers$ **do**
  $comms.add(Q3(c))$
end for

Q8 Enumerate all the communities with their sizes.

$nc \leftarrow rank_{B_s}(1, |B_s|)$
for $i \leftarrow 1$ **to** nc **do**
  $ss \leftarrow select_{B_s}(1, i+1) - select_{B_s}(1, i)$
  $sc \leftarrow select_{B_c}(1, i+1) - select_{B_c}(1, i)$
  $size\_list.add(ss + sc)$
end for

Q9 Enumerate all the communities with their densities.

$nc \leftarrow rank_{B_s}(1, |B_s|)$
for $i = 1$ **to** nc **do**
  $sources \leftarrow select_{B_s}(1, i+1) - select_{B_s}(1, i)$
  $centers \leftarrow select_{B_c}(1, i+1) - select_{B_c}(1, i)$
  $edges \leftarrow sources \cdot centers$
  $nodes \leftarrow sources + centers$
  $densities.add(\frac{edges}{nodes \cdot (nodes-1)/2})$
end for

4.4 Compact representation of $\mathcal{R}$

We define a sequence of integers A and two bitmaps $B1$ and $B2$ for representing $\mathcal{R}(V_R, E_R)$. Sequence A is defined as $A = a(1)...a(i)...a(N)$, where $|A| = |E_R|$, $a(i)$ is the i-th nonempty direct adjacency list of $\mathcal{R}$, and N is the total number of vertices with at least one edge in $\mathcal{R}$. Bitmap $B1$ is $10^{|a(1)|-1}...10^{|a(N)|-1}$, so $|B1| = |E_R|$. $B2$ is a bitmap such that $B2[i] = 1$ iff vertex i does not have out-neighbors and $|B2| = |V_R|$. Figure 5 (b) shows an example. The space using WTs is $|A|(H_0(A)+1)+|\sigma|+o(|A|\log\sigma)$, where $\sigma = |V_R|$. We answer neighbor queries on $\mathcal{R}$ as follows:

out-neighbors (u)
if $access_{B2}(u) = 0$ **then**
  $x \leftarrow rank_{B2}(0, u)$
  $start \leftarrow select_{B1}(1, x)$
  $end \leftarrow select_{B1}(1, x+1) - 1$
  for $i \leftarrow start$ **to** end **do**
    $out.add(access_A(i))$
  end for
end if

in-neighbors (u)
$occur \leftarrow rank_A(u, |A|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_A(u, i)$
  $p \leftarrow rank_{B1}(1, y)$
  $in.add(select_{B2}(0, p))$
end for

Query	Time complexity
Q1/Q2	$O(\|output\| \cdot \log \sigma)$
Q3/Q4	$O((\|output\| + 1) \log \sigma)$
Q5	$O(\log \sigma)$
Q6	$O(\|output\| \log \sigma))$
Q7	$O(\|output\| \log \sigma) \ldots O(\|Q1\| \cdot \|output\| \log \sigma)$
Q8/Q9	$O(\|output\|)$

Table 7: Time complexity for community queries.

On directed graphs, in-neighbors and out-neighbors are found in time $O((|output|+1)\log\sigma)$. On undirected graphs we choose arbitrarily to represent each edge $\{u, v\}$ as (u, v) or (v, u). Consequently, finding the neighbors of a node requires carrying out both algorithms.

To carry out out/in-queries on the whole graph, we must query $\mathcal{H}$ and $\mathcal{R}$ (for *UBP* or *DBP* partitions) or $\mathcal{H}_u$, $\mathcal{H}_d$ and $\mathcal{R}_d$ (for *UDBP* partitions), and merge the results. Community queries are carried out only on $\mathcal{H}$ (or $\mathcal{H}_u$ and $\mathcal{H}_d$ for *UDBP*, then merging the results). Our pseudocodes on X and B addressed the directed case. Those for communities representing undirected graphs are very easy to derive, and left as an exercise.

5. EXPERIMENTAL EVALUATION

We evaluate our compact data structures supporting out/in-neighbor and community queries. We are interested in space/time requirements in terms of the community graph $\mathcal{H}$ (or $\mathcal{H}_u + \mathcal{H}_d$), the remaining graph $\mathcal{R}$, and the complete graph $G = \mathcal{H} \cup \mathcal{R}$.

5.1 Space/time evaluation

We compare space/time efficiency using the representations below. We refer as WT-N-b to representing sequence X with wavelet trees and bitmaps with RG [18], and as WT-N-r to using wavelet trees for X and bitmaps compressed with RRR [28]. N is the sampling parameter used for bitmap implementations (if left as a variable, it gives a space/time tradeoff). We will not give the results for using GMR [17] on $\mathcal{H}$ because the space achieved is not competitive.

T4 WT-N-b ($\mathcal{H}$) + Re-Pair GMR ($\mathcal{R}$)
T5 WT-N-r ($\mathcal{H}$) + Re-Pair GMR ($\mathcal{R}$)
T6 WT-N-b ($\mathcal{H}$) + k2tree ($\mathcal{R}$)
T7 WT-N-r ($\mathcal{H}$) + k2tree ($\mathcal{R}$)
T8 WT-N-b ($\mathcal{H}$) + WT-64-b ($\mathcal{R}$)
T9 WT-N-b ($\mathcal{H}$) + WT-64-r ($\mathcal{R}$)
T10 WT-N-r ($\mathcal{H}$) + WT-64-b ($\mathcal{R}$)
T11 WT-N-r ($\mathcal{H}$) + WT-64-r ($\mathcal{R}$)

We use the definitions of Section 4.2 and our Web graphs and social networks of Table 1. We use *UBP* to represent

Data Set	$\mathcal{H}_u$	$\mathcal{H}_d$	$\mathcal{R}$	%
Facebook	457,968	-	359,122	43.95
Flickr	8,009,958	7,697,030	6,906,992	30.50
LiveJournal	24,270,703	17,374,268	27,078,099	39.24
eu-2005	-	17,568,086	1,667,054	8.66
uk-2002	-	270,951,713	27,162,049	9.11
ar-2005	-	602,662,165	37,337,293	5.83

Table 8: Component sizes and % of edges not participating in communities.

the (undirected) Facebook graph, *DBP* on Web graphs, and *UDBP* for Flickr and LiveJournal graphs. Table 8 shows the number of edges for components $\mathcal{H}_u$ (= $\mathcal{H}$ on *UBP*), $\mathcal{H}_d$ (= $\mathcal{H}$ on *DBP*), and $\mathcal{R}$, and the percentage of $\mathcal{R}$ (the part not forming communities) with respect to the original graph. We also evaluated Web graphs with *UDBP*, and Flickr and LiveJournal with *UBP*, but these were less competitive.

Techniques T4–T11 support community queries on $\mathcal{H}$ and out/in-neighbor queries on $\mathcal{H}+\mathcal{R}$. We measure compression on G by computing $bpe = \frac{bits(\mathcal{H})+bits(\mathcal{R})}{edges(\mathcal{H})+edges(\mathcal{R})}$ and access time $query\_time = query\_time(\mathcal{H}) + query\_time(\mathcal{R})$. Table 9 shows the bpe for $\mathcal{H}_d$ representations of Web graphs, $\mathcal{H}_u$ representation of Facebook, and $\mathcal{H}_u$ plus $\mathcal{H}_d$ for Flickr and LiveJournal graphs. We compute $bpe(\mathcal{H}) = \frac{bits(\mathcal{H})}{edges(\mathcal{H})}$. When using *UDBP* we show $bpe(\mathcal{H}_u)$ and $bpe(\mathcal{H}_d)$ separately. We observe higher compression on Web graphs than on social networks and better results using compressed bitmaps (WT-r). We also show time efficiency for neighbor and different community queries in Table 10 (for *UDBP* the times for $\mathcal{H}_u$ and $\mathcal{H}_d$ must be added together). As it can be seen, *all community queries are supported within a few microseconds.*

Figure 6 shows the space and time on Web graphs and social networks, considering both partitions $\mathcal{H}$ (or $\mathcal{H}_u+\mathcal{H}_d$) and $\mathcal{R}$, and out/in-neighbor queries. On Web graphs, we include the results using k2tree (T2) and VNM + k2tree (T3), recall Figure 2. While in general T2 and T3 dominate the space/time tradeoff, variant T7 (WTs on $\mathcal{H}$ and k2tree on $\mathcal{R}$) is competitive in some cases. Nevertheless, we remind that T4–T11 additionally support community queries.

On social networks, on the other hand, we achieve better results using techniques T4–T11 than using techniques T1–T3 (we include T2, the best performing technique of Figure 3, in Figure 6). Variant T11 provides the least space, whereas variants T9 and T4 provide other relevant space/time tradeoffs. T2 is only mildly relevant on Facebook. There exist other proposals in the literature achieving less space [26], but these do not handle community queries.

5.2 Discussion

Figure 7 shows the histogram of community sizes found in each dataset, and Figure 8 shows the density-normalized distribution. We observe that community sizes on Web graphs are larger than on social networks, which explains the better compression of $\mathcal{H}$ for those graphs.

In Figure 8, we observe that the maximum community density (Def. 2) is 0.6. This is obtained when $|S| = 2$ and $|C| = 3$ or vice versa. We observe that social networks have more of those tiny communities. Second, it can be easily seen from Def. 2 that in a large community where $|S| = |C|$, the maximum density is 0.5. Figure 8 shows that the number of communities with density 0.5 is greater on Web graphs than on social networks. These density observations also explain

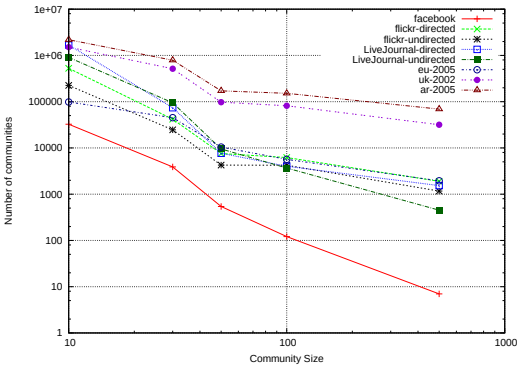

Figure 7: Community size histogram.

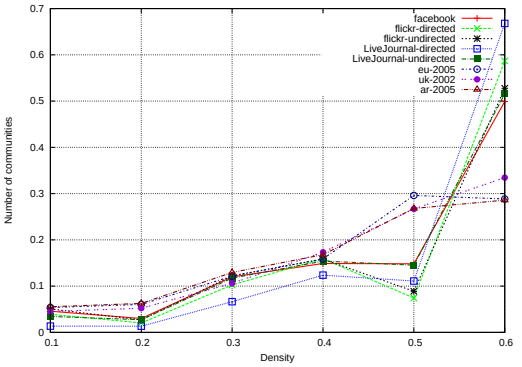

Figure 8: Community density normalized distribution.

why we obtain better compression of $\mathcal{H}$ for those graphs.

When compressing Web graphs with out-neighbor support, we found that it is possible to combine compression schemes that take advantage of different graph properties. Reducing edge redundancy works well with methods that use node ordering to exploit locality and similarity. This combination, however, does not work well on social networks. The reason is that on social networks community sizes are smaller, and then representing the virtual nodes adds a considerable overhead. In the context of compression techniques that support out/in-neighbor queries, the reduction of edges on Web graphs still allows k2tree to exploit sparseness of the adjacency matrix, but on social networks it removes part of the sparseness and degrades the compression.

Our biclique representation of $\mathcal{H}$, instead, does not represent those virtual nodes. This is why it works particularly well on the small communities arising in social networks.

6. CONCLUSIONS

This work has three contributions. First, it offers improved space/time tradeoffs for out-neighbor queries on Web graphs, including the best space requirements reported up to date. We achieve this by combining a technique that reduces the number of graph edges (VNM) [11] with techniques that reorder nodes to exploit ordering, locality, similarity and interval/integer encoding (AD [2] and WebGraph [5, 4]). Second, it achieves improved space/time tradeoffs and the best reported space for supporting out/in-neighbor queries on Web graphs. To achieve this result, we combine VNM with a technique that exploits the sparseness of the adjacency matrix (k2tree) [8]. Third, it proposes a novel

Compression	eu-2005	uk-2002	ar-2005	Facebook	Flickr Directed	Flickr Undirected	LiveJournal Directed	LiveJournal Undirected
WT-32-b	3.82	3.55	3.20	12.11	14.26	13.15	16.91	15.40
WT-32-r	2.77	2.52	2.12	11.72	13.35	12.09	16.15	15.00

Table 9: Compression (bpe) of $\mathcal{H}$ on Web graphs and social networks using compact data structures.

Queries	eu-2005	uk-2002	ar-2005	Facebook	Flickr Directed	Flickr Undirected	LiveJournal Directed	LiveJournal Undirected
Out WT-32-b	6.70	8.71	8.62	6.49	8.91	7.28	12.69	10.16
Out WT-32-r	9.66	12.02	12.03	9.17	12.94	10.45	18.22	14.39
In WT-32-b	6.84	8.62	8.68	8.11	10.15	10.00	13.01	13.25
In WT-32-r	9.44	11.51	11.89	11.50	14.88	14.25	18.88	18.93
Q3 WT-32-b	3.15	6.07	5.12	1.06	0.89	1.04	2.56	2.47
Q3 WT-32-r	3.53	6.87	6.05	1.28	0.98	1.09	2.77	2.59
Q4 WT-32-b	1.99	3.69	2.42	0.85	0.84	0.82	2.12	2.25
Q4 WT-32-r	2.39	4.40	3.04	0.90	0.94	0.96	2.59	2.71
Q5 WT-32-b	2.22	4.10	2.78	1.22	1.40	1.39	3.21	3.27
Q5 WT-32-r	2.61	4.86	3.43	1.31	1.61	1.61	3.68	3.80
Q6 WT-32-b	7.17	8.69	10.14	4.76	5.63	10.24	8.81	7.85
Q6 WT-32-r	8.83	11.79	13.18	7.14	9.01	10.20	11.75	10.46
Q7 WT-32-b	119.21	147.88	233.56	39.22	98.76	162.91	74.71	58.24
Q7 WT-32-r	163.02	199.47	303.11	53.48	114.33	189.35	90.70	78.95

Table 10: Times for community queries ($usecs$) on Web graphs and social networks representing $\mathcal{H}$ with compact data structures.

compressed structure that enables community and out/in-neighbor queries on Web graphs and social networks. This is the first compressed structure that supports community and out/in-neighbor queries.

7. ACKNOWLEDGEMENTS

This work is partially funded by Fondecyt Grant 1-110066.

8. REFERENCES

[1] C. Aggarwal and H. Wang. *Managing and Mining Graph Data*. Springer, 2010.

[2] A. Apostolico and G. Drovandi. Graph compression by bfs. *Algorithms*, 2(3):1031–1044, 2009.

[3] L. Becchetti, C. Castillo, D. Donato, R. A. Baeza-Yates, and S. Leonardi. Link analysis for Web spam detection. *ACM Transactions on the Web*, 2008.

[4] P. Boldi, M. Santini, and S. Vigna. Permuting Web graphs. In *WAW*, pages 116–126, 2009.

[5] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004.

[6] P. Boldi and S. Vigna. The Webgraph framework II: Codes for the World-Wide Web. In *DCC*, page 528, 2004.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[8] N. Brisaboa, S. Ladra, and G. Navarro. K2-trees for compact Web graph representation. In *SPIRE*, LNCS 5721, pages 18–30, 2009.

[9] A. Z. Broder. Min-wise independent permutations: Theory and practice. In *ICALP*, page 808, 2000.

[10] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the Web. *Computer Networks*, 33(1-6):309–320, 2000.

[11] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to Web graph compression with communities. In *WSDM*, pages 95–106, 2008.

[12] M. Cha, A. Mislove, and P. K. Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *WWW*, pages 721–730, 2009.

[13] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009.

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

[15] F. Claude and G. Navarro. A fast and compact Web graph representation. In *SPIRE*, pages 118–129, 2007.

[16] F. Claude and G. Navarro. Extended compact Web graph representations. In *Algorithms and Applications (Ukkonen Festschrift)*, LNCS 6060, pages 77–91, 2010.

[17] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, pages 368–373, 2006.

[18] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Posters WEA*, pages 27–38, 2005.

[19] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[20] C. Karande, K. Chellapilla, and R. Andersen. Speeding up algorithms on compressed web graphs. In *WSDM*, pages 272–281, 2009.

[21] M. Katarzyna, K. Przemyslaw, and B. Piotr. User position measures in social networks. In *SNA-KDD*, pages 1–9, 2009.

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

[23] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the Web for emerging cyber-communities. *Computer Networks*, 31(11-16):1481–1493, 1999.

[24] S. Ladra. *Algorithms and Compressed Data Structures for Information Retrieval*. PhD thesis, University of A

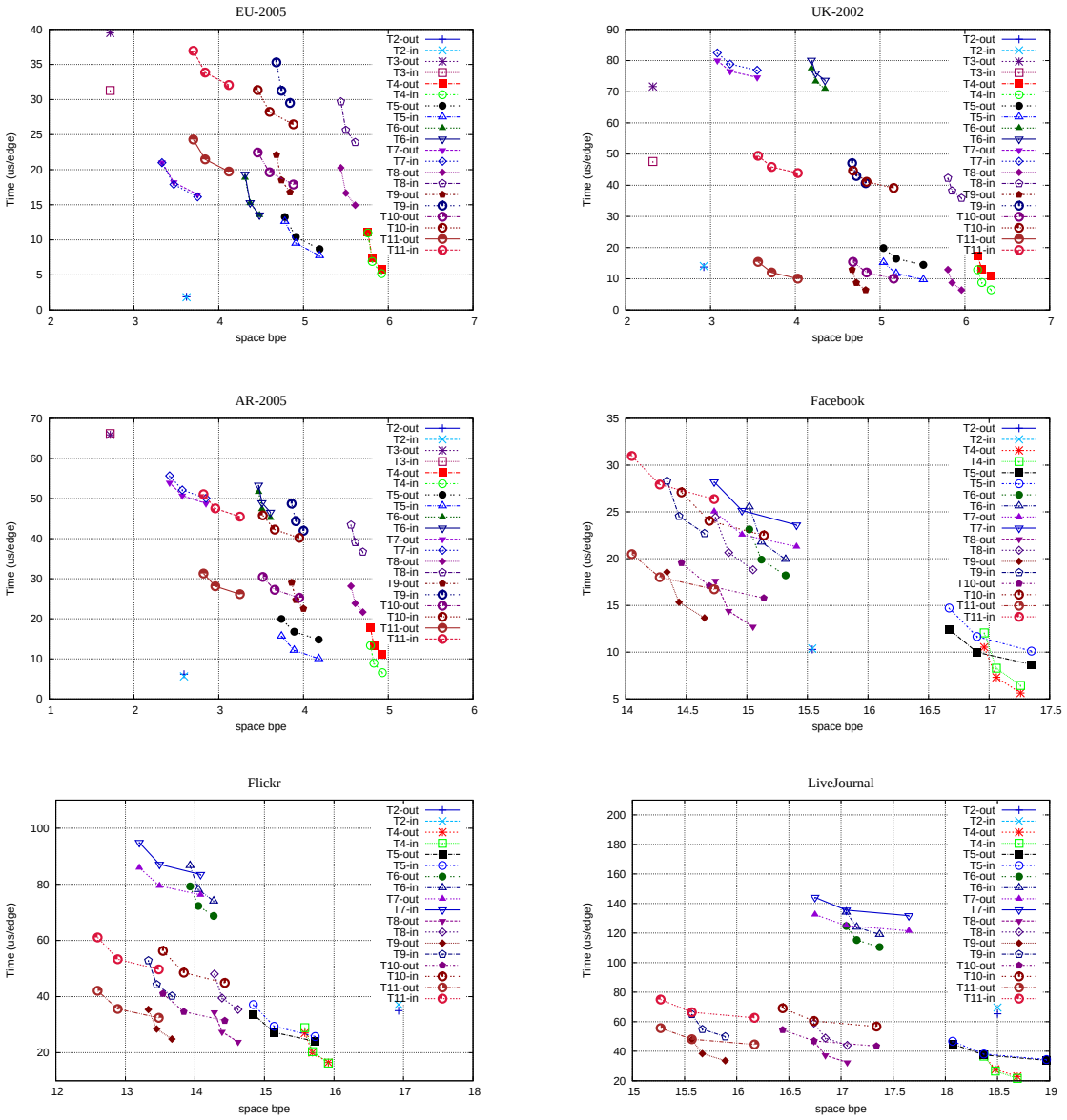

Figure 6: Space/time efficiency with out/in-neighbors queries on $\mathcal{H}+\mathcal{R}$ for representations T4–T11.

Coruña, Spain, 2011.

[25] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *DCC*, pages 296–305, 1999.

[26] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *KDD*, pages 533–542, 2010.

[27] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, 2007.

[28] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.

[29] K. H. Randall, R. Stata, J. L. Wiener, and R. Wickremesinghe. The link database: Fast access to graphs of the Web. In *DCC*, pages 122–131, 2002.

[30] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms (s). In *AIRWeb*, 2007.

[31] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[32] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *DCC*, pages 213–222, 2001.

[33] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *WOSN*, 2009.