Compressed Representation of Web and Social Networks via Dense Subgraphs *

Cecilia Hernández[1,2] and Gonzalo Navarro[2]

[1] Dept. of Computer Science, University of Concepción, Chile,
[2] Dept. of Computer Science, University of Chile, Chile,
{chernand,gnavarro}@dcc.uchile

Abstract. Mining and analyzing large web and social networks are challenging tasks in terms of storage and information access. In order to address this problem, several works have proposed compressing large graphs allowing neighbor access over their compressed representations. In this paper, we propose a novel compressed structure aiming to reduce storage and support efficient navigation over web and social graph compressed representations. Our approach uses clustering and mining for finding dense subgraphs and represents them using compact data structures. We perform experiments using a wide range of web and social networks and compare our results with the best known techniques. Our results show that we improve the state of the art space/time tradeoffs for supporting neighbor queries. Our compressed structure also enables mining queries based on dense subgraphs, such as cliques and bicliques.

Keywords: Web graphs, Social networks, Compressed data structures

1 Introduction

A wide range of information is available both in the link structure of web graphs and in the relation structure of social networks. The link structure is usually used for ranking algorithms such as PageRank [6] and HITS [18], whereas the relation structure of social networks is used for mining and analysis tasks such as identifying interest groups and understanding information propagation [22]. Many of these tasks are based on graph algorithms that often rely on having the complete graph in main memory. This imposes a great demand on system resources, especially on the current growth rate of these graphs. For instance, the indexed web contains about 50 billion pages[1], and Facebook has more than 800 million users world-wide[2]. This continous growth has pushed the search for compressed representations with efficient storage space and access times.

We aim for effective techniques to store and access large graphs to benefit both web graph and social networks. Finding regularities in graphs has shown

* Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F.

[1] www.worldwidewebsize.com

[2] www.facebook.com/press/info.php?statistics

to be successful for defining compressed structures. For instance, Boldi and Vigna [5] exploit similarity of adjacency lists and locality of reference of nearby pages using URL ordering for nodes; Brisaboa et al. [7] exploit sparseness and clustering of the adjacency matrix; Buehrer and Chellapilla [9] exploit complete bipartite subgraphs (bicliques) on web graphs, that is, groups of pages that share the same outlinks. Combining clustering with node ordering together with similarity and locality [2] have improved space/time tradeoffs.

In the context of social networks, URL node ordering does not apply. Instead, deriving a node ordering from some clustering seems good for compressing those networks [2]. Maserrat and Pei [21] use the idea of decomposing the graph into small dense subgraphs, which can be represented more space-efficiently. Hernández and Navarro [17] use a related idea, detecting bicliques and representing them compactly.

In this paper we further pursue the line of representing web and social graphs by detecting dense subgraphs and representing them compactly. We generalize from previous successful experiences that rely on finding cliques [21] and bicliques [9, 17] and adapt clustering algorithms to find broader constructions that lie in between. More precisely, we consider a *dense subgraph* as a pair (S, C) of subsets of nodes, such that every node in S points to every node in C, but where S and C need not be disjoint. The case $S = C$ corresponds to cliques and the case of disjoint sets corresponds to bicliques. We show that these more general dense subgraphs appear sufficiently more often than cliques and bicliques, thus it pays off to design a more general compact representation for them. We design a representation that efficiently solves out/in-neighbor queries in symmetric form, that is, it not only offers the basic functionality of an adjacency list, but it also allows one to determine which nodes point to a given node. In addition, the structure lets us easily obtain the dense subgraphs it has found on the compression process, which is useful for mining activities.

Our experimental results show that our new technique offers the best space and time performance on undirected social networks, or on directed social networks where we need to retrieve both out- and in-neighbors. In many cases our structures use less space than the best alternatives that can only retrieve out-neighbors. On web graphs we achieve the best spaces under the last assumption, yet our times are higher than other alternatives.

2 Related Work

Randall et al. [24] first proposed lexicographic ordering of URLs as a way to exploit locality (i.e., that pages tend to have hyperlinks to other pages on the same domain) and similarity of (nearby) adjacency lists for compressing Web graphs. Later, Boldi and Vigna [5] proposed the WebGraph framework. This approach exploits power-law distributions, similarity and locality using URL ordering.

On a later work, Boldi et al. [3] explored and evaluated existing and novel node ordering methods, such as URL, lexicographic, Gray ordering. More re-

cently, Boldi et al. [2] designed node orderings based on clustering methods, and achieved improvements on compressing web graphs and social network using clustering based on Layered Label Propagation (LLP). A different and competitive node ordering was proposed by Apostolico and Drovandi [1]. Their approach orders the nodes based on a Breadth First Traversal (BFS) of the graph, and then used their own encoding. Buehrer and Chellapilla [9] exploit the existence of many groups consisting of sets of pages that share the same outlinks, which defines complete bipartite subgraphs (bicliques). Their approach is based on reducing the number of edges by defining virtual nodes that are artificially added in the graph to connect the two sets in a biclique. They apply this process iteratively on the graph until the edge reduction gain is no longer significant. A grammar-based approach called Re-Pair also reduces edges [20, 13]. Re-Pair consists of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol. Hernández and Navarro [17] explored more in general the idea of combining an edge-reduction method with a subsequent encoding of the resulting graph. They obtained the best results, improving upon the state of the art, by combining Buehrer and Chellapilla's [9] bicliques with Apostolico and Drovandi's [1] encoding of the graph.

Brisaboa et al. [7] exploit the sparseness and clustering of the adjacency matrix to reduce space while providing out/in-neighbor navigation in a natural symmetric form, in a structure called *k2tree*. The k2tree scheme 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). It divides the adjacency matrix into k^2 submatrices of size n^2/k^2. Completely empty subzones are represented just with a 0-bit, whereas nonempty subzones are marked with a 1-bit and recursively subdivided. The leaf nodes contain the actual bits of the adjacency matrix, in compressed form. Recently, Claude and Ladra [12] improved the compression performance for web graphs by combining k2tree with the Re-Pair-based representation [13].

There exist recent works on compressing social networks [10, 21]. The approach by Chierichetti et al. [10] is based on the Webgraph framework [5], using shingling ordering [8] and exploiting link reciprocity. Even though they provide interesting compression for social networks, the scheme requires decompressing the graph in order to retrieve out-neighbors. Maserrat and Pei [21] scheme achieves compression by defining a Eulerian data structure using multi-position linearization of directed graphs. This scheme is based on decomposing the graph into small dense subgraphs and supports out/in-neighbor queries in sublinear time. Claude and Ladra [12] improve upon this scheme by combining it with the use of compact data structures. Hernández and Navarro [17] use Buehrer and Chellapilla's [9] technique to find bicliques only (cliques and other patterns were not supported) and represent these using compact data structures.

3 Dense Subgraph Patterns

We represent a web graph as a directed graph $G = (V, E)$ where V is a set of vertices (pages) and $E \subseteq V \times V$ is a set of edges (hyperlinks). For an edge $e=(u,v)$,

we call u the *source* and v the *center* of e. In social networks, nodes are individuals (or other types or agents) and edges represent some relationship between the two nodes. These graphs can be directed or undirected. We make undirected graphs directed by representing reciprocal edges. In this case, retrieving the out- plus the in-neighbors of a node gives its neighbors in the undirected graph. Thus from now on we consider only directed graphs.

For technical reasons that will be clear next, we also consider that the directed graphs contain all the edges of the form (u, u). Most web and social graphs do not contain any such edge. In that case we simply omit reporting those edges when our representations retrieve them. If there are graphs containing some such edges, one can indicate with a small bitmap of $|V|$ bits which nodes u have a self-loop, and remove the spurious edges when a query retrieves them. We will find patterns of the following kind.

Definition 1. A *dense subgraph* $H(S, C)$ of $G = (V, E)$ is a graph $G'(S \cup C, S \times C)$, where $S, C \subseteq V$.

Note this includes in particular cliques ($S = C$) and bicliques ($S \cap C = \emptyset$). Our goal will be to represent the $|S| \cdot |C|$ edges of a dense subgraph $H(S, C)$ in space proportional to $|S| + |C| - |S \cap C|$. Thus the bigger the dense subgraphs we detect, the more space we save at representing their edges.

3.1 Discovering dense subgraphs

In order to discover dense subgraphs, we apply iteratively clustering and mining passes until the number of subgraphs discovered in a pass is below a threshold. In each pass we look for subgraphs over certain size (measured in number of edges, $|S| \cdot |C|$), and decrease this threshold for the next passes. The goal is to avoid that extracting a small dense subgraph precludes the identification of a larger dense subgraph, which gives a higher benefit.

As even finding a clique of a certain size within a graph is NP-complete, we need to use fast heuristics on these huge graphs. We first improve the scalable clustering algorithm based on shingles, proposed by Buehrer and Chellapilla [9]. Once the clustering has identified nodes whose adjacency lists are sufficiently similar, we run a heavier frequent itemset mining algorithm [9] inside each cluster. The algorithms proposed by Buehrer and Chellapilla [9] were designed to find only bicliques. To make the algorithms sensitive to dense subgraphs we insert all the edges $\{(u, u), u \in V\}$ in E, as anticipated. This is sufficient to make the clustering and mining algorithms find more general dense subgraphs. As explained, the spurious edges added are removed at query time.

The clustering algorithm represents each adjacency list with P fingerprints (hash values), generating a matrix of fingerprints of $|V|$ rows and P columns (we used $P = 2$). Then it traverses the matrix column-wise. At stage i the matrix rows are sorted lexicographically by their first i column values, and the algorithm groups the rows with the same fingerprints in columns 1 to i. When the number of rows in a group falls below a threshold, it is converted into a cluster formed by the nodes corresponding to the rows. Groups that remain after the last column is processed are also converted into clusters.

On each cluster we apply the frequent itemset mining algorithm, which extracts dense subgraphs from the cluster. This algorithm first computes frequencies of the nodes mentioned in the adjacency lists, and sorts the list by decreasing frequency of the nodes. Then the nodes are sorted lexicographically according to their lists. Now each list is inserted into a prefix tree, discarding nodes of frequency 1. Each node p in the prefix tree has a label (consisting of the node id), and it represents the sequence $l(p)$ of labels from the root to the node. Such node p also stores the range of graph nodes whose list start with $l(p)$.

Note that a tree node p at depth $c = |l(p)|$ representing a range of s graph nodes identifies a dense subgraph $H(S, C)$, where S are the graph nodes in the range stored at the tree node, and C are the graph nodes listed in $l(p)$. Thus $|S| = s$ and $|C| = c$. We can thus point out all the tree nodes p where $s \cdot c$ is over the size threshold, and choose them from largest to lowest saving (which must be recalculated each time we choose the largest).

The execution time for discovering dense subgraphs is about 0.1 ms per link. Our construction is not yet optimized, however.

4 A Compact Representation

After we have extracted all the interesting dense subgraphs from $G(V, E)$, we represent G as the set of dense subgraphs plus a *remaining* graph.

Definition 2. Let $G(V, E)$ be a directed graph, and let $H(S_r, C_r)$ be edge-disjoint dense subgraphs of G. Then the corresponding *dense subgraph* representation of G is $(\mathcal{H}, \mathcal{R})$, where $\mathcal{H} = \{H(S_1, C_1), \ldots, H(S_N, C_N)\}$ and $\mathcal{R} = G - \bigcup H(S_r, C_r)$ is the remaining graph.

4.1 Compact sequence representations

Many compact data structures use as a basic tool a bitmap supporting rank, select, and 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 $B[1, i]$. The 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 [11] and good implementations are available (e.g. RG [15]). Later, Raman et al. (RRR) [23] managed to compress the bitmap while retaining constant query times. The space becomes $nH_0(B) + o(n)$ bits, where $H_0(B)$ is the zero-order entropy of 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. A representation (GMR) [4] 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. The wavelet tree (WT) [16] 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. As our alphabets will be very large, we use the version called "without pointers" [14], which does not involve an extra space of the form $O(\sigma \log n)$.

4.2 Compact representation of $\mathcal{H}$

Let $\mathcal{H} = \{H_1, \ldots, H_N\}$ be the dense subgraph collection found in the graph, based on Definition 2. We represent $\mathcal{H}$ as a sequence of integers X with a corresponding bitmap B. Sequence $X = X_1 : X_2 : \ldots : X_N$ represents the sequence of dense subgraphs and bitmap $B = B_1 : B_2 : \ldots B_N$ is used to mark the separation between each subgraph. We now describe how a given X_r and B_r represent the dense subgraph $H_r = H(S_r, C_r)$.

We define X_r and B_r based on the overlapping between the sets S and C. Sequence X_r will have three components: L, M, and R, written one after the other in this order. Component L lists the elements of $S - C$. Component M lists the elements of $S \cap C$. Finally, component R lists the elements of $C - S$. Bitmap $B_r = 10^{|L|}10^{|M|}10^{|R|}$ gives alignment information to determine the limits of the components. In this way, we avoid repeating nodes in the intersection, and have sufficient information to determine all the edges of the dense subgraph. Table 1 describes the construction algorithm of X and B.

Figure 1 illustrates our approach using a single subgraph as an example of graph. (a) shows a typical graph representation, (b) our graph representation, and (c) our compact representation.

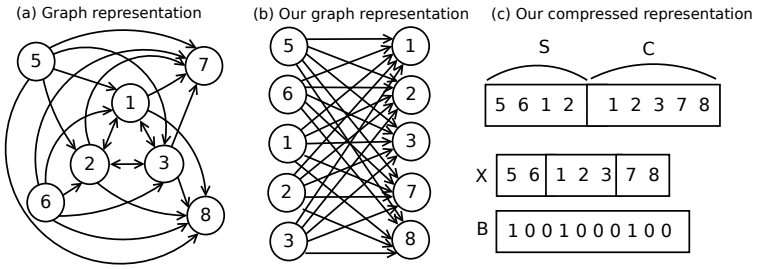

Fig. 1. Subgraph example with our compact representation.

We compress the graph $G = \mathcal{H} \cup \mathcal{R}$, using sequence X and bitmap B for $\mathcal{H}$. For $\mathcal{R}$ we use some existing compression schemes for graphs.

To support our query algorithms, X and B are represented with compact data structures for sequences that implement rank/select/access operations. We use WTs [16] for sequence X and a compressed bitmap representation (RRR [23]) for bitmap B. The total space is $|X|(H_0(X) + 1) + o(|X| \log \sigma) + |X|H_0(B) + o(|X|)$, where $\sigma \leq |V|$ is the number of vertices in subgraph $\mathcal{H}$. The $|X|H_0(X) + o(|X| \lg \sigma)$ owes to the wavelet tree representation, whereas $|X|H_0(B) + o(|X|)$

owes to the bitmap B. Note that $|X|$ is the sum of the number of nodes of the dense subgraphs in $\mathcal{H}$, which is much less than the number of edges in the subgraph it represents.

We answer out/in-neighbor queries as described by the algorithms in Table 1. Their complexity is $O((|output|+1)\log\sigma)$, which is away from optimal by a factor $O(\log\sigma)$. To exemplify the treatment of (u,u) edges, these algorithms always remove them before delivering the query results (as explained, more complex management is necessary if the graph actually contains some of those edges). Note this finds only the edges represented in component $\mathcal{H}$; those in $\mathcal{R}$ must be also extracted, using the out/in-neighbor algorithm provided by the representation we have chosen for it. The out-neighbor algorithm works as follows. Using $select_u(X,i)$ we find all the places where node u is mentioned in X. This corresponds to some X_r, but we do not now where. Then we analyze B to determine whether this occurrence of u is inside component L, M, or R. In cases L and M, we use B again to delimit components M and R, and output all the nodes of X_r in those components. If u is in component R, instead, there is nothing to output. The in-neighbors is analogous.

Build X and B
Input: S, C in A (patterns)
Output: X and B
$X = \varepsilon$
$B = \varepsilon$
for $i = 0$ **to** $A.N$ **do**
$S = A[i].S$
$C = A[i].C$
$L = S - C$
$M = S \cap C$
$R = C - S$
$X = X : L : M : R$
$B = B : 10^{|L|}10^{|M|}10^{|R|}$
end for
return X, B

Find Out-neighbors
Input: X, B and vertex u
Output: Out-neighbors of u
$out = \varepsilon$
$occur = rank_X(u, |X|)$
for $i = 1$ **to** $occur$ **do**
$y = select_X(u, i)$
$p = select_B(0, y+1)$
$o = p - y$ { $= rank_B(1, p)$ }
$m = o \bmod 3$
if $m = 1$ **then**
$s = select_B(1, o+1) - o$
$e = select_B(1, o+3) - o - 3$
else if $m = 2$ **then**
$s = select_B(1, o) - o + 1$
$e = select_B(1, o+2) - o - 2$
else
$s = 1$
$e = 0$
end if
for $j = s$ **to** e **do**
$d = access_X(j)$
if $d \neq u$ **then**
$out = out : d$
end if
end for
end for
return out

Find In-neighbors
Input: X, B and vertex u
Output: In-neighbors of u
$in = \varepsilon$
$occur = rank_X(u, |X|)$
for $i = 1$ **to** $occur$ **do**
$y = select_X(u, i)$
$p = select_B(0, y+1)$
$o = p - y$ { $= rank_B(1, p)$ }
$m = o \bmod 3$
if $m = 2$ **then**
$s = select_B(1, o-1) - o + 2$
$e = select_B(1, o+1) - o - 1$
else if $m = 0$ **then**
$s = select_B(1, o-2) - o + 3$
$e = select_B(1, o) - o$
else
$s = 1$
$e = 0$
end if
for $j = s$ **to** e **do**
$d = access_X(j)$
if $d \neq u$ **then**
$in = in : d$
end if
end for
end for
return in

Table 1. Algorithms for building and querying $\mathcal{H}$.

An interesting advantage of our compressed structure is that it enables the retrieval of the actual dense subgraphs found on the graph. For instance, we are able to recover cliques and bicliques in addition to navigating the graph. This information can be useful for mining and analyzing web and social graphs. The time complexity is $O(|output| \cdot \log\sigma)$. Note that cliques can be found explicitly when $L = R = 0$ and $M! = 0$ or when cliques are included in dense subgraphs.

5 Experimental Evaluation

We implemented the algorithms for dense subgraph discovery, encoding, and querying in C++. We used Linux PC with 16 processors Intel Xeon at 2.4GHz, with 72 GB of RAM and 12 MB of cache. We used WT, RG, GMR, and RRR implementations from the compact structures library *libcds* (`libcds.recoded.cl`). For WT we used the variant "without pointers".

We considered four compressed representations for web graphs and social networks. One is WebGraph (BV), version 3.0.1, corresponding to their last implementation that uses LLP ordering [2]. This reaches the lowest space within the WebGraph framework. A second one is Claude and Ladra's implementation of MP_k [12], an improvement upon the proposal of Maserrat and Pei [21] for social networks. Third, we use the k2tree implementation [7], with the last improvements [19]. Finally, we use the k2partitioning [12], which combines k2tree and Re-Pair on web graphs. We use these implementations both for compressing our $\mathcal{R}$ component, and to compare our results with the state of the art.

We experiment with the web graphs and social networks displayed in Figure 2, which are available at `law.dsi.unimi.it` by the WebGraph framework project. We use the natural order for the web graph data sets. In addition, we use the LiveJournal (directed graph) data set, available from the SNAP project (Stanford Network Analysis Package, `snap.stanford.edu/data`).

5.1 Compression performance

We first study the performance of our dense subgraph mining algorithm, and its impact on compression performance. Our dense subgraph discovery algorithm has two parameters: *edge_saving*, defining the minimum subgraph size of interest, and *thres*, the minimum number of subgraphs to find on the current pass before looking for smaller dense subgraphs. We use these two parameters to iteratively discover dense subgraphs from larger to smaller sizes. We use $edge\_saving = 500, 100, 50, 30, 15, 6$ and $thres = 10$ for small graphs and 500 for larger graphs.

We meausure the percentage of the edges that are captured by the dense subgraph mining algorithm. Figure 2 shows that more than 90% of the web graphs can be represented using dense subgraphs. The percentage is much lower, around 50%, on social graphs, with the exception of Hollywood-2011 (HW2011). This anticipates the well-known fact that web graphs compress much better than social graphs. The table also compares the result if we detect only bicliques [17]($\mathcal{H}(1)$), and if we detect more general dense subgraphs ($\mathcal{H}(2)$). Even though, there is not much difference (percentage) in finding bicliques or dense subgraphs, the space/time efficiency is improved using the later as seen in Figures 3(c),(d).

Figure 2 compares the compression we achieve with the alternatives we have chosen. We show the *edge_saving* (ES) value used for discovering dense subgraphs, the ratio $RE/|X|$, where $|X| = \sum_r |S_r| + |C_r| - |S_r \cap C_r|$ and $RE = \sum_r |S_r| \cdot |C_r|$. The compression performance in bpe obtained on web and social graphs, where $bpe = \frac{bits(\mathcal{H})+bits(\mathcal{R})}{edges(\mathcal{H})+edges(\mathcal{R})}$. We use wavelet trees without

pointers and compressed bitmaps (RRR) for compressing $\mathcal{H}$. For compressing $\mathcal{R}$, we use k2tree for web graphs and MP_k for social networks (with *enron* as an exception, where using k2tree on $\mathcal{R}$ provides better compression than MP_k as seen in Figure 2). We do not show results of using MP_k for web graphs because it did not provide competitive compression and we include k2Partitioning in Figure 3. We observe that the ratio $RE/|X|$ is higher on web graphs than on social networks, and consequently provides better compression. For the alternatives we consider BV and k2tree on web graphs, and BV and MP_k on social graphs. For BV we use parameters $m = 3$ and $w = 7$ (as recomended by their authors); for MP_k we use the best k.

We note that BV is unable to retrieve in-neighbors. To carry out a fair comparison, we follow BV authors suggestion [2] for supporting out-in/neighbor queries. They suggest to compute the set E_{sym} of all symetric edges, that is, those for which both (u, v) and (v, u) exist. Then they consider the graph $G_{sym} = (V, E_{sym})$ and $G_d(V, E - E_{sym})$, so that storing G_{sym}, G_d, and the transpose of G_d enables both queries. The space we report for BV considers this arrangement. Nevertheless, we show the space for BV(1) (including offsets for random access), which is the basic representation that supports out-neighbors only.

The results show that our technique improves upon the space achieved by the best alternative techniques, sometimes by a wide margin, in case we are interested in solving both out/in-neighbor queries. We recall that in the case of undirected graphs one is forced anyway to be able to report out/in-neighbors, or what is the same for BV, to represent the whole graph and its transpose. We remark that in some cases (including all of the Web graphs) our technique uses even less space than the BV representation of the original graph, which does not support in-neighbor queries (Figure 2 BV(1) column).

We note that part of our graph ($\mathcal{R}$) is indeed represented with the best alternative technique (k2tree for web graphs, MP_k for social graphs), but we are able to improve the compression on the part, $\mathcal{H}$, that can be represented with dense subgraphs.

5.2 Space/time performance

We now compare the combined space/time efficiency of our approach against the state-of-the-art techniques. Recall that BV is adapted to support out/in-neighbor queries. In all cases we will only report out-neighbor times. In-neighbor times were also measured, but they are very similar to out-neighbor times for all the data structures, and showing them just clutters the plots.

Our structure compresses $G = \mathcal{H} \cup \mathcal{R}$ using compact data structures for $\mathcal{H}$, and uses k2tree for compressing $\mathcal{R}$ on web graphs, and k2tree and MP_k on social networks. We denote WT-b our technique applying wavelet tree without pointers for sequence X and RG on the wavelet tree bitmaps, and WT-r as applying wavelet tree without pointers and RRR on the wavelet tree bitmaps. We compare Wavelet tree (WT) with Golynski (GMR) [4] for sequence X (Figures 3(c),(d)). Bitmap B also uses RG or RRR depending on the notation. The sampling parameter for both bitmap representations (RG or RRR) is 16, 32, and

64, which yields a line for each technique. Note that our time is always the sum of the time spent on $\mathcal{H}$ and on $\mathcal{R}$.

Finally, it can be seen that our technique improves upon MP_k [12] in space and time. We must insist in that this does not diminish the merit of MP_k structure: we do use it to represent our $\mathcal{R}$ part. The point is that this structure improves in space and time when combined with ours. Figures 3(a),(b) compare our results for dblp-2011 and LJSNAP with MP_k, since the k2tree and BV are not competitive.

Figures 3(c),(d),(e),(f) show space/time efficiency on web graphs. We show in Figure 3(c),(d) the space/time efficiency of web graphs represented by $\mathcal{H}$ only, where we are able to capture more than 91% of the total of edges in Eu-2005 and over 94% in Arabic-2005. We compare space/time performance between representing dense subgraphs (DS) and bicliques (BI) only. We remark that our representation enables neighbor and mining queries and it is more efficient in space and time than the representation of the complete graph, where we add the representation of $\mathcal{R}$ using k2tree. In Figures 3(e),(f) we compare our results for representation $G = \mathcal{H} \cup \mathcal{R}$ against k2tree, k2partitioning and BV. We substantially improve BV compression, although with significantly higher access time. We also improve the compression of k2tree and k2partitioning, again using more time (between three and four times slower than k2tree).

We remind that our compressed structure also enables to recover cliques, bicliques and other dense subgraphs, while none of the alternatives easily support these mining queries.

6 Conclusions

This paper proposes a novel compressed structure for web and social graphs based on extracting dense subgraphs and representing them with compact data structures. We generalize previous biclique discovery algorithms so that they detect dense subgraphs, and our experiments show that this generalization pays off in terms of compression performance. As extracting them is non-trivial, these dense subgraphs may be useful for other graph mining and analysis purposes, and our representation gives easy access to them. Our compressed data structure avoids redundancy both in the node and in the edge representation of dense subgraphs, and supports efficient out-neighbor and in-neighbor queries, with symmetric techniques and similar performance.

Our results show that we are able to improve both space and neighbor retrieval time, with respect to the best current alternatives, on social networks. On web graphs we improve the best previous space, yet our times are significantly higher. These comparisons assume that either the graphs are undirected (as some social networks) or that we want to be able to retrieve both out- and in-neighbors (separately). However, on some web graphs our space is even better than the best alternatives that only support out-neighbor queries.

					$G = \mathcal{H} \cup \mathcal{R}$			k2tree	BV	BV(1)
Data set	Nodes	Edges	$\mathcal{H}(1)$	$\mathcal{H}(2)$	ES	RE/$\|X\|$	bpe	bpe	bpe	bpe
eu-2005	862,664	19,235,140	91.30	91.86	6	7.29	2.67	3.45	7.19	4.20
in-2004	7,414,866	194,109,311	93.29	94.51	6	14.17	1.49	1.73	3.37	1.78
uk-2002	18,520,486	298,113,762	90.80	91.41	6	8.50	2.52	2.78	5.59	2.81
arabic-2005	22,744,080	639,999,458	94.16	94.61	6	11.56	1.85	2.47	4.02	2.17
it-2004	41,291,594	1,150,725,436	93.24	94.34	6	12.40	1.79	1.77	3.94	2.15
enron	69,244	276,143	46.28	48.47	6	2.06	10.07	10.31	18.30	7.26
								MP_k		
enron	69,244	276,143	46.28	48.47	6	2.06	15.42	17.02	18.30	7.26
dblp-2011	986,324	6,707,236	49.88	65.51	100	8.38	8.41	8.48	10.13	10.13
LJSNAP	4,847,571	68,993,773	53.77	56.37	500	12.66	13.02	13.25	23.16	16.07
LJ2008	5,363,260	79,023,142	54.17	56.51	100	4.88	13.04	13.35	17.84	11.84
HW2011	2,180,759	228,985,632	92.68	94.34	500	8.53	4.05	4.17	5.23	5.23

Fig. 2. Graph properties and compression performance for random access.

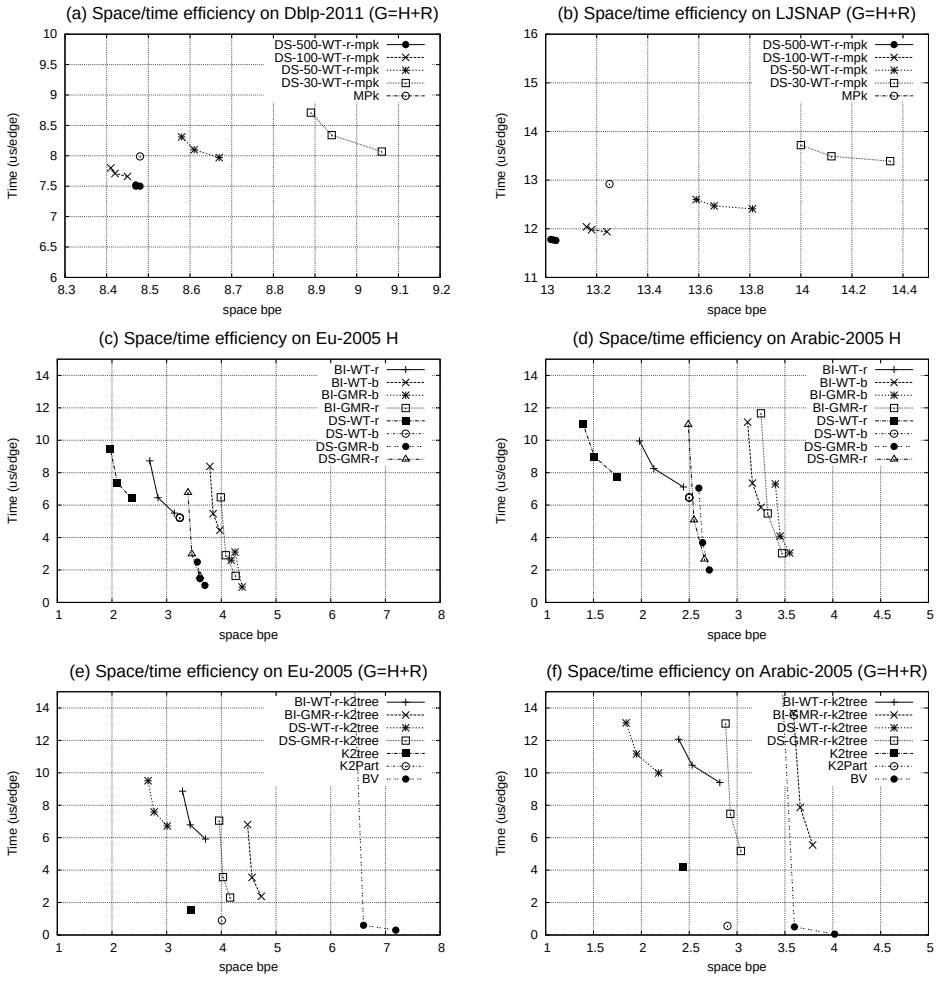

Fig. 3. Space/time efficiency on web and social graphs with outneighbor queries.

References

1. A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.
2. P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pp. 587–596, 2011.
3. P. Boldi, M. Santini, and S. Vigna. Permuting Web graphs. In *WAW*, pp. 116–126, 2009.
4. A. Golynski, I. Munro and S. Srinivasa. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, pp. 368-373, 2006.
5. P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pp. 595–602, 2004.
6. S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Comp. Netw.*, 30(1-7):107–117, 1998.
7. N. Brisaboa, S. Ladra, and G. Navarro. K2-trees for compact Web graph representation. In *SPIRE*, pp. 18–30, 2009.
8. A. Z. Broder. Min-wise independent permutations: Theory and practice. In *ICALP*, page 808, 2000.
9. G. Buehrer and K. Chellapilla. A scalable pattern mining approach to Web graph compression with communities. In *WSDM*, pp. 95–106, 2008.
10. F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, pp. 219–228, 2009.
11. D. Clark. Compact pat trees. In *Ph.D. Thesis, University of Waterloo*, 1996.
12. F. Claude and S. Ladra. Practical representations for web and social graphs. In *CIKM*, pp. 1185–1190, 2011.
13. F. Claude and G. Navarro. Fast and compact Web graph representations. *ACM TWEB* 4(4):art.16, 2010.
14. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *SPIRE*, pp. 176–187, 2008.
15. R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *WEA*, posters, pp. 27–38, 2005.
16. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pp. 841–850, 2003.
17. C. Hernández and G. Navarro. Compression of Web and social graphs supporting neighbor and community queries. *SNA-KDD*, 2011.
18. J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *JACM*, 46(5):604–632, 1999.
19. S. Ladra. *Algorithms and compressed data structures for information retrieval.* Ph.D. Thesis, University of A. Coruña, Spain, 2011.
20. N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *DCC*, pp. 296–305, 1999.
21. H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *KDD*, pp. 533–542, 2010.
22. A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Internet Measurement Conference*, pp. 29–42, 2007.
23. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pp. 233–242, 2002.
24. K. H. Randall, R. Stata, J. L. Wiener, and R. Wickremesinghe. The link database: Fast access to graphs of the Web. In *DCC*, pp. 122–131, 2002.