Knowl Inf Syst (2014) 40:279–313
DOI 10.1007/s10115-013-0648-4

REGULAR PAPER

Compressed representations for web and social graphs

Cecilia Hernández · Gonzalo Navarro

Received: 16 August 2012 / Revised: 10 April 2013 / Accepted: 15 April 2013 /
Published online: 26 April 2013

Abstract Compressed representations have become effective to store and access large Web and social graphs, in order to support various graph querying and mining tasks. The existing representations exploit various typical patterns in those networks and provide basic navigation support. In this paper, we obtain unprecedented results by finding "dense subgraph" patterns and combining them with techniques such as node orderings and compact data structures. On those representations, we support out-neighbor and out/in-neighbor queries, as well as mining queries based on the dense subgraphs. First, we propose a compression scheme for Web graphs that reduces edges by representing dense subgraphs with "virtual nodes"; over this scheme, we apply node orderings and other compression techniques. With this approach, we match the best current compression ratios that support out-neighbor queries (i.e., nodes pointed from a given node), using 1.0–1.8 bits per edge (bpe) on large Web graphs, and retrieving each neighbor of a node in 0.6–1.0 microseconds (μs). When supporting both out- and in-neighbor queries, instead, our technique generally offers the best time when using little space. If the reduced graph, instead, is represented with a compact data structure that supports bidirectional navigation, we obtain the most compact Web graph representations (0.9–1.5 bpe) that support out/in-neighbor navigation; yet, the time per neighbor extracted raises to around 5–20 μs. We also propose a compact data structure that represents dense subgraphs without using virtual nodes. It allows us to recover out/in-neighbors and answer other more complex queries on the dense subgraphs identified. This structure is not competitive on Web graphs, but on social networks, it achieves 4–13 bpe and 8–12 μs per out/in-neighbor retrieved, which improves upon all existing representations.

Partial versions of this article appeared in *Proc. SNA-KDD 2011* and *Proc. SPIRE 2012*.

C. Hernández (✉)
Computer Science Department, University of Concepción, Concepción, Chile
e-mail: cecihernandez@udec.cl; chernand@dcc.uchile.cl

C. Hernández · G. Navarro
Computer Science Department, University of Chile, Santiago, Chile
e-mail: gnavarro@dcc.uchile.cl

Keywords Compressed data structures · Graph mining · Web graphs · Social networks

1 Introduction

Web graphs represent the link structure of the Web. They are usually modeled as directed graphs where nodes represent pages and edges represent links among pages. On the other hand, social networks represent relationships among social entities. These networks are modeled by undirected or directed graphs depending on the relation they model. For instance, the friendship relation in Facebook is symmetric and, then, it is modeled by an undirected graph, whereas the "following" relation on Twitter and LiveJournal is not symmetric, and therefore, it is modeled by a directed graph.

The link structure of Web graphs is often used by ranking algorithms such as PageRank [10] and HITS [38], as well as for spam detection [6,50], for detecting communities [27, 39], and for understanding the structure and evolution of the network [26,27]. A social network structure is often used for mining and analysis purposes, such as identifying interest groups or communities, detecting important actors [51,57], and understanding information propagation [17,37,45]. Those algorithms use a graph representation that supports at least forward navigation (i.e., to the out-neighbors of a node or those pointed from it), and many require backward navigation as well (i.e., to the in-neighbors of a node or those that point to it).

Managing and processing these graphs are challenging tasks because Web graphs and social networks are growing in size very fast. For instance, a recent estimation of the indexable Web size states that it is over 7.8 billion pages (and thus, around 200 billion edges),[1] and Facebook has over 950 million active users worldwide.[2] Google has recently augmented the user search experience by introducing the *knowledge graph*,[3] which models the relationship of about half-million entities over 3.5 billion relationships among the entities. This *knowledge graph* is used in addition to the Web graph to improve the search efficacy.

Different approaches have been used to manage large graphs. For instance, streaming and semi-streaming techniques can be applied with the goal of processing the graph sequentially, ideally in one pass, although a few passes are allowed. The idea is to use main memory efficiently, avoiding random access to disk [25]. External memory algorithms define memory layouts that are suitable for graph algorithms, where the goal is to exploit locality in order to reduce I/O costs, reducing random accesses to disk [56]. Another approach is the use of distributed systems, where distributed memory is aggregated to process the graph [53]. However, depending on the problem, the synchronization and communication required may impose I/O costs similar to those of the external memory approach.

Compressed data structures aim to reduce the amount of memory use by representing graphs in compressed form while being able to answer the queries of interest without decompression. Even though these compressed structures are usually slower than uncompressed representations, they are still much faster than incurring I/O costs: They can be orders of magnitude faster when they can fit completely in main memory graphs that would otherwise require disk storage. When considering a distributed scenario, they allow the graphs to be deployed on fewer machines, yielding important savings in communication costs and energy.

[1] www.worldwidewebsize.com, on August 6, 2012.

[2] http://newsroom.fb.com/content/default.aspx?NewsAreaId=22, considering June 2012.

[3] http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html.

Several proposals use compressed data structures for Web graphs, mainly enabling out-neighbor queries [4,7,21,32]; yet, some also support bidirectional navigation (i.e., handle out/in-neighbor queries) [11,20]. Some more recent ones address social networks [9,19,23,43].

In this paper, we introduce new approaches to develop competitive compressed data structures for managing and processing large Web and social graphs. The main contributions of this work follow:

- We enhance an existing technique to detect bicliques [16] so that it detects more general "dense subgraphs." These include cliques, bicliques, and in general not necessarily disjoint pairs of node sets where all in the first set point to all in the second set.[4] We study the effectiveness of the technique and demonstrate that it captures a fair amount of the structure of Web graphs (more than 90 %) and social networks (around 60 %), improving upon the detection of bicliques (where the sets must be disjoint). We show how to process large graphs in the secondary memory. This new graph mining technique is key to the success of the compressed representations we develop.
- We apply their "virtual node mining" technique [16] on the discovered dense subgraphs, which replaces the edges of the dense subgraph by a virtual node with fewer links. We then list the nodes in the BFS order of Apostolico and Drovandi [4] and use their encoding. The result is a Web graph representation with out-neighbor query support that is either very close to or better than, in space and time, the best current representation [32]: On large Web graphs, it uses 1.0–1.8 bits per edge (bpe) and retrieves each neighbor in 0.6–1.0 microsecond (μs). We show, however, that our technique is more robust as it performs equally well on the transposed Web graph, whereas the one by Grabowski and Bieniecki [32] performs significantly worse.
- By maintaining the BFS ordering after virtual node mining, but now using a bidirectional representation (*k2-tree*) on the resulting graph [11], we obtain the smallest existing representation with out/in-neighbor support: 0.9–1.5 bpe, much smaller than in the previous item. The price is that the query time is higher: 5–20 μs per extracted neighbor.
- We design a novel compressed data structure to represent the dense subgraphs that does not use virtual nodes. This representation supports not only out/in-neighbor navigation, but also various graph mining queries based on the dense subgraphs discovered, such as listing cliques and bicliques, retrieving density and size of the subgraphs, finding node participation in different subgraph patterns, and so on. While this technique is not competitive with the previous one on Web graphs (yet, it supports other queries), it excels in social networks, where it achieves the best spaces so far with support for out/in-neighbor queries: 4–13 bpe and 8–12 μs per retrieved neighbor.

Conference versions of this work appeared in *SNA-KDD* workshop [35] and in *SPIRE* [36]. This article extends that work with a thorough analysis of the quality of the dense subgraph finding algorithm, a secondary memory variant of the algorithm, its application to the transposed Web graphs, improved combinations of the scheme with BFS orderings, and the study of other graph mining queries.

In all the experiments we described in this paper, we used a Linux PC with 16 processors Intel Xeon at 2.4 GHz, with 72 GB of RAM, and 12 MB of cache. We used `g++` compiler with full optimization.

[4] The term "dense subgraph" appears in the literature with different meanings [41], but in this paper, we use it to mean the described generalization of cliques and bicliques.

2 Related work

We divide this section in two parts. First, we survey compression techniques for Web and social graphs, and the supported queries. Second, we discuss compact data structures based on bitmaps and symbol sequences that provide guarantees in terms of space and access times. Such structures are the basis for the compressed data structure we present in Sect. 5.

2.1 Compressed representations for Web and social graphs

Compressing Web graphs has been an active research area for some time. Suel and Yuan [52] built a tool for Web graph compression distinguishing global links (pages on different hosts) from local ones (pages on the same host) and combining different coding techniques, such as Huffman and Golomb codes. Adler and Mitzenmacher [1] achieved compression by using similarity. The idea was to code an adjacency list by referring to an already coded adjacency list of another node that points to many of the same pages. They used this idea with Huffman coding to achieve compression of global links. Randall et al. [48] 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 [7] proposed the WebGraph framework. This approach also exploits power-law distributions, similarity and locality using URL node ordering. Essentially, given a node ordering that enhances locality and similarity of nearby lists, WebGraph uses an encoding based on gaps and pointers to near-copies that takes advantage of those properties. The main parameters of this compression technique are w and m, where w is the window size and m is the maximum reference count. The window size means that the list l_i can only be expressed as a near-copy of l_{i-w} to l_{i-1}, whereas the reference count of list l_i is $r(l_i) = 0$ if it is not expressed as a near-copy of another list, or $r(l_i) = r(l_j) + 1$ if l_i is encoded as a near-copy of list l_j. Increasing w and m improves compression ratio, but also increases access time.

In a later work, Boldi et al. [8] explored existing and novel node ordering methods, such as URL, lexicographic, Gray ordering, etc. More recently, Boldi et al. [9] designed node orderings based on the clustering methods and achieved improvements on compressing Web graphs and social networks with a clustering algorithm called layered label propagation (LLP). A different and very competitive node ordering was proposed by Apostolico and Drovandi [4]. Their approach orders the nodes based on a breadth-first traversal (BFS) of the graph, and then, they used their own encoding that takes advantage of BFS. They encode the out-degrees 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. They achieve compression by dividing those lists into chunks and taking advantage of locality and similarity. The compression scheme works on chunks of l nodes. Parameter l (called the level) provides a tradeoff between compression performance and time to retrieve the adjacency list of a node.

Buehrer and Chellapilla [16] exploited 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 applied this process iteratively on the graph until the edge reduction gain is no longer significant. Then, they applied delta codes on the edge-reduced graph. However, they did not report times for extracting neighbors. They called this scheme as virtual node mining (VNM). Anh and Moffat [3] also exploit similarity and locality of adjacency lists, but they divide the lists into groups of h consecutive lists. A *model* for a group is built as a union of the group lists. They reduced lists by replacing consecutive sequences in all h lists by a new symbol. The process can be made recursive by applying it to the n/h representative lists. They finally applied codes such as ς-codes [7] over all lists. This approach is somehow similar to that of Buehrer and Chellapilla [16], but Anh and Moffat[3] do not specify how they actually detect similar consecutive lists.

Grabowski and Bieniecki [32] (see also [31]) recently provide a very compact and fast technique for Web graphs. Their algorithms are based on blocks consisting of multiple adjacency lists in a way similar to Anh and Moffat work [3], reducing edge redundancy, but they use a compact stream of flags to reconstruct the original lists. Their encoding is basically a reversible merge of all lists. The parameter h sets the number of adjacency lists stored in blocks. Increasing the value of h improves compression rate at the cost of access time.

Another approach that can also be seen as decreasing the number of total edges and adding virtual nodes was proposed by Claude and Navarro [21]. This approach is based on Re-Pair [40], a grammar-based compressor. Re-Pair repeatedly finds the most frequent pair of symbols in a sequence of integers and replaces it with a new symbol.

Most of the Web graph compression schemes (as the ones described above) support out-neighbor queries, that is, the list of nodes pointed from a given node, just as an adjacency list. Being able to solve in-neighbor queries (i.e., the list of nodes pointing to a given node) is interesting for many applications from random sampling of graphs to various types of mining and structure discovery activities, as mentioned in Sect. 1. It is also interesting in order to represent undirected graphs without having to store each edge twice.

Brisaboa et al. [11] exploited the sparseness and clustering of the adjacency matrix to reduce space while providing out/in-neighbor navigation in a natural symmetric form, using a structure called *k2tree*. They have recently improved their results by applying BFS node ordering on the graph before building the *k2tree* [12]. This achieves the best known space/time tradeoffs supporting out/in-neighbor access for Web graphs. 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. Complete 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 [23] improved the compression performance on Web graphs by combining the *k2tree* with the Re-Pair-based representation [21]. Another representation able to solve out/in-neighbors [20] was obtained by combining the Re-Pair-based representation [21] with compact sequence representations [22] of the resulting adjacency lists. The times for out- and in-neighbor queries are not symmetric.

Some recent works on compressing social networks [19,43] have unveiled compression opportunities as well, although in much less degree than on Web graphs. The approach by Chierichetti et al. [19] is based on the Webgraph framework [7], using shingling ordering (based on Jaccard coefficient) [13,28] and exploiting link reciprocity. Even though they achieve interesting compression for social networks, their approach requires decompressing the graph in order to retrieve the out-neighbors. Maserrat and Pei [43] achieve compression by defining an 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 [23] improve upon this scheme by combining it with the use of compact data structures.

2.2 Compact data structures for sequences

We make use of compact data structures based on bitmaps (sequences of bits) and sequences of symbols. These sequences support operations $rank$, $select$ and $access$. 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 ith 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 [24], and good implementations are available (e.g., RG [29]). Later, Raman et al. [49] 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, $H_0(B) = \frac{n_0}{n} \log \frac{n}{n_0} + \frac{n_1}{n} \log \frac{n}{n_1} \le 1$, where B has n_0 zeros and n_1 ones (we use binary logarithms by default). Good implementations are also available (i.e., RRR [22]).

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) [33] 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 $H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c} \le \log \sigma$, n_c being the number of occurrences of c in S. As our alphabets will be very large, we use the version "without pointers" [22], which saves an extra space of the form $O(\sigma \log n)$. Another sequence representation (GMR) [30] uses $n \log \sigma + n\, o(\log \sigma)$ bits and supports $rank$ and $access$ in time $O(\log \log \sigma)$, and $select$ in $O(1)$ time.

3 Dense subgraphs

In this section, we describe the algorithm to discover dense subgraphs, such as bicliques, cliques, and generalizations, and study the quality of our algorithm. This technique is the basis for all the compressed representations that follow.

3.1 Basic notions

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 of agents) and edges represent some relationship between the two nodes. These graphs can be directed or undirected. In case they are undirected, we make them directed by representing both reciprocal directed edges. Thus, from now on we consider only directed graphs.

We follow the idea of "dense communities" in the Web described by Kumar et al. [39] and Dourisboure et al. [27], where a community is defined as a group of pages related to a common interest. Such Web communities are characterized by dense-directed bipartite subgraphs. In fact, Kumar et al. [39] summarize that a "random large enough and dense bipartite subgraph of the Web almost surely has a core (a complete bipartite subgraph)", which they aim to detect. Left sets of dense subgraphs are called *Fans*, and right sets are called *Centers*. In this work, we call the sets *Sources* (S) and *Centers* (C), respectively, which is the same naming given by Buehrer and Chellapilla [16]. One important difference of our work from Kumar et al. [39] and Dourisboure et al. [27] is that we do not remove edges before applying the discovery algorithm. In contrast, both works [27,39] remove all *nepotistic links*, that is, links between two pages that belong to the same domain. In addition, Dourisboure et al. [27] removes *isolated* pages, that is, pages with zero out-neighbors and in-neighbors.

For technical reasons that will be clear next, we will add all the edges (u, u) to our directed graphs. We use a small bitmap of |V| bits to mark which nodes u actually had a self-loop. We use this bitmap to remove the spurious self-loops from the edges output by our structures.

We also note that the discovery algorithms are applied over Web graphs with *natural node ordering* [9], which is basically URL ordering, because they provide better results than using other node orderings.

We will find patterns of the following kind.

Definition 3.1 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 that, Definition 3.1 includes cliques ($S = C$) and bicliques ($S \cap C = \emptyset$), but also more general subgraphs. Our goal is to represent the $|S| \cdot |C|$ edges of a dense subgraph using $O(|S| + |C|)$ space. Two different techniques to do so are explored in Sects. 4 and 5.

3.2 Discovering dense subgraphs

In this section, we describe how we discover dense subgraphs. Even finding a clique of a certain size is NP-complete, and the existing algorithms require time exponential on that size (e.g., Algorithm 457 [15]). Thus, we need to resort to fast heuristics for our huge graphs of interest. Besides, we want to capture other types of dense subgraphs, not just cliques. We first use a scalable clustering algorithm [16], which uses the idea of "shingles" [28]. Once the clustering has identified nodes whose adjacency lists are sufficiently similar, we run a heavier frequent itemset mining algorithm [16] inside each cluster. This mining algorithm is the one that finds sets of nodes S that point to all the elements of another set of nodes C (they can also point to other nodes).

This algorithm was designed to find bicliques: A node u cannot be in S and C unless (u, u) is an edge. As those edges are rare in Web graphs and social networks, this algorithm misses the opportunity to detect dense subgraphs and is restricted to find bicliques.

To make the algorithm sensitive to dense subgraphs, we insert all the edges $\{(u, u), u \in V\}$ in E, as anticipated. This is sufficient to make the frequent itemset mining algorithm find the more general dense subgraphs. The spurious edges added are removed at query time, as explained.

The clustering algorithm represents each adjacency list with P fingerprints (hash values), generating a matrix of fingerprints of $|V|$ rows and P columns. 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 small number, 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 discovers 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. This prefix tree has a structure similar to the tree obtained by the hierarchical termset clustering [47]. Each node p in the prefix tree has a

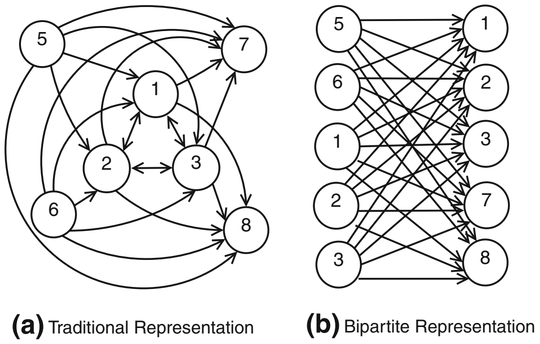


Fig. 1 Dense subgraph representation

label (consisting of the node id), and it represents the sequence $l(p)$ of labels from the root to the node. Such node p stores also 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 is the graph nodes in the range stored at the tree node, and C is 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).

Figure 1a shows a dense subgraph pattern with the traditional representation, and Fig. 1b shows the way we represent them using the discovery algorithm described.

The whole algorithm can be summarized in the following steps. Figure 2 shows an example.

Step 1 Clustering-1 (*build hashed matrix representing* G) We traverse the graph specified as set of adjacency lists, adding edges (u, u). Then, we compute a hash value H associated with each edge of the adjacency list P times and choose the P smallest hashes associated with each adjacency list. Therefore, for each adjacency list, we obtain P hash values. This step requires $O(P|E|)$ time.
Step 2 Clustering-2 (*build clusters*) We build clusters consisting of groups of similar hashes, by sorting the hash matrix by columns, and select adjacency lists associated with clusters based on hashes. This requires $O(P|V|\log|V|)$ time.
Step 3 Mining-1 (*reorder cluster edges*) We compute edge frequencies on each cluster, sorting them from largest to smallest (discarding edges with frequency of 1), and reorder them based on that order. This step takes $O(|E|\log|E|)$ time.
Step 4 Mining-2 (*discover dense subgraphs and replacing*) We compute a prefix tree for each cluster, with tree nodes labeled with the node id of edges. Dense subgraphs $(G'(S \cup C, S \times C))$ with higher edge saving $(|S| \times |C|)$ are identified in the tree. The overall step is bounded to $O(|E|\log|E|)$ time.

Therefore, the overall algorithm time complexity, taking P as a constant, is bounded by $O(|E|\log|E|)$.

In Sect. 4, the dense subgraphs found $H(S, C)$ will be replaced by a new virtual node whose in-neighbors are S and whose out-neighbors are C. As the result is still a graph, the dense subgraph discovery process can be repeated on the resulting graph. In Sect. 5, instead, the graph $H(S, C)$ will be extracted only from the original graph and represented using a compact data structure.

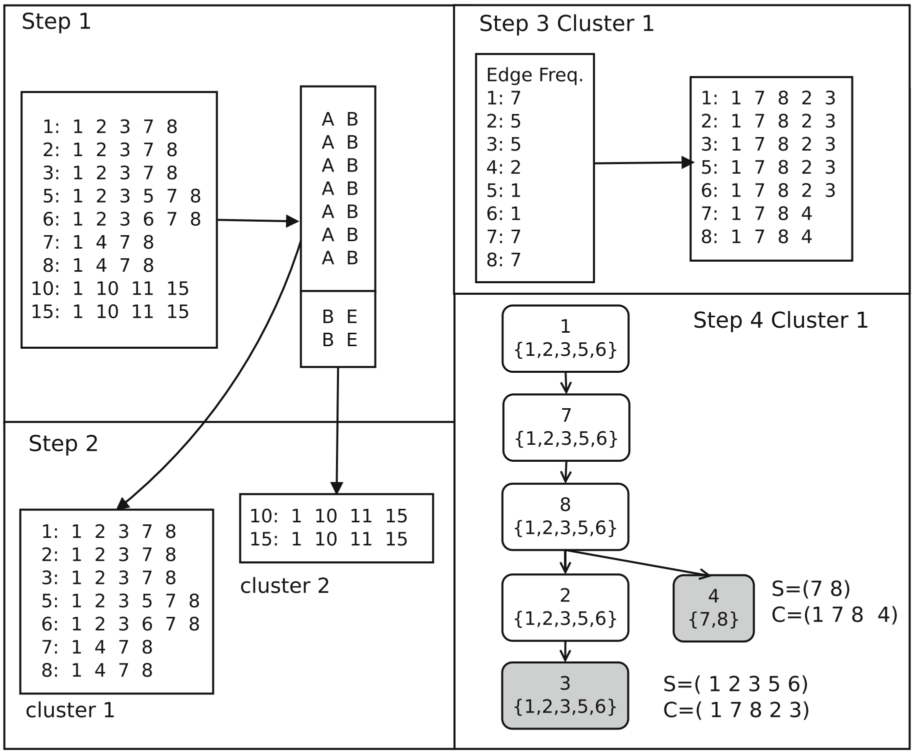


Fig. 2 Example of the dense subgraph discovery process

Table 1 Compression metrics using different *P* values with eu-2005

P	# Cliques	|Cliques|	# Bicliques	Edges	Nodes	Ratio
2	33,482	248,964	58,467	17,208,908	2,357,455	7.30
4	34,237	246,022	60,226	17,199,357	2,426,753	7.08
8	34,863	245,848	60,934	17,205,357	2,524,240	6.81

3.3 Evaluation of the discovery algorithm

First, we evaluate the sensibility of the number of hashes (parameter P) used in the first step of our clustering. For doing so, we use a real Web graph (eu-2005, see Table 7). We measure the impact of P in various metrics that predict compression effectiveness. Table 1 shows the number of discovered cliques (# Cliques), total number of edges in those cliques (|Cliques|), number of bicliques (# Bicliques), total number of edges in cliques and bicliques (Edges), total number of nodes participating in cliques and bicliques (Nodes), and the ratio between both (Ratio, which gives the reduction factor using our technique of Sect. 5). All these metrics show that using $P = 2$ is slightly better than using other values. When increasing P, the algorithm discovers more and smaller cliques and bicliques, but the overall compression in terms of representing more edges with fewer vertices is better with $P = 2$.

Table 2 Synthetic clique graphs with different number of nodes (Nodes), edges (Edges), maximum clique size (MC), and total number of vertices participating in cliques (R)

Name	Nodes	Edges	d	MC	R	avg size
PL	999,993	9,994,044	9.99	0	0	–
V16	65,536	610,500	9.31	15	6,548	9.5
V16	65,536	1,276,810	19.48	30	3,785	17.09
V16	65,536	2,161,482	32.98	50	2,398	27.21
V16	65,536	4,329,790	66.06	100	1,263	51.83
V17	131,072	1,214,986	9.26	15	13,130	9.48
V17	131,072	2,542,586	19.39	30	7,589	17.05
V17	131,072	4,309,368	32.87	50	4,790	27.23
V17	131,072	8,739,056	66.67	100	2,495	52.95
V20	1,048,576	9,730,142	9.76	15	104,861	9.50
V20	1,048,576	20,293,364	19.60	30	60,822	17.02
V20	1,048,576	34,344,134	32.90	50	38,544	27.07
V20	1,048,576	69,324,658	66.18	100	20,102	52.10

Column d gives the average number of edges per node, and the last column is the average clique size

Second, we evaluate our subgraph discovery algorithm. For doing so, we use the *GTgraph* suite of synthetic graph simulators.[5] From this suite, we use the `SSCA#2` generator to create random-sized clique graphs [5, 18]. We use the parameter *MaxCliqueSize* to set the maximum size of cliques (MC), set the *Scale* parameter to 16, 17, or 20, so as to define 2^{16}, 2^{17} or 2^{20} vertices on the graph, and set the parameter *ProbIntercliqueEdges* = 0.0 (which tells the generator to create a clique graph, that is, a graph consisting of isolated cliques). Therefore, with this generator, we can control precisely the actual cliques present in the graph, and their corresponding sizes. We call those *real cliques*.

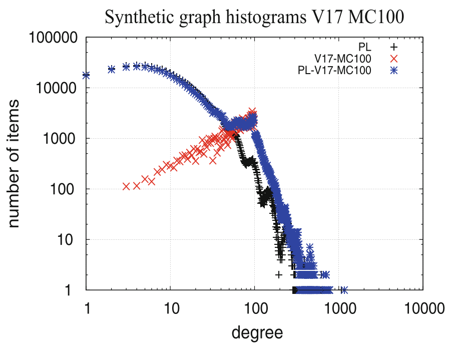
We also use the generator `R-MAT` of the suite to create a power-law graph without any cliques. The properties of the synthetic clique graphs and the power-law graph used are described in Table 2. The first graph, PL, is the power-law graph, whereas the others are clique graphs (V16,V17,V20). Finally, we define new graphs (PL-V16, PL-V17, and PL-V20), which are the result of merging graphs PL with V16, PL with V17, and PL with V20. The merging process is done by computing the union of the edge sets belonging to the PL graph and one of the clique graphs. That is, both PL and Vxx share the same set of nodes (called 1 to $|V|$), and we take the union of the edges in both graphs. We apply our dense graph discovery algorithm on those merged graphs, whose features are displayed in Table 3. Figure 3 (left) shows the out-degree histogram for PL, V17 (with $MC = 100$), and PL-V17 graphs. We evaluate the ability of our discovery algorithm to extract all the real cliques from these graphs.

For evaluation purposes, we also use MCL (Markov Cluster Process), a clustering algorithm [54] (and later mathematically analyzed [55]), which has been mostly applied in bioinformatic applications [14], but also in social network analysis [44]. MCL simulates a flow, alternating matrix expansion and matrix inflation, where expansion means taking the power of a matrix using the matrix product, and inflation means taking the Hadamard power followed by a diagonal scaling. MCL deals with both labeled and unlabeled graphs, while the

[5] Available at www.cse.psu.edu/~madduri/software/GTgraph.

Table 3 Synthetic merged power-law and clique graphs

Name	Nodes	Edges	MC	d
PL-V16	999,993	10,604,408	15	10.6
PL-V16	999,993	11,270,660	30	11.27
PL-V16	999,993	12,155,249	50	12.15
PL-V16	999,993	14,323,320	100	14.32
PL-V17	999,993	11,208,968	15	11.20
PL-V17	999,993	12,536,277	30	12.53
PL-V17	999,993	14,303,175	50	14.30
PL-V17	999,993	18,732,584	100	18.73
PL-V20	1,048,576	19,724,071	15	18.81
PL-V20	1,048,576	30,287,168	30	28.88
PL-V20	1,048,576	44,337,825	50	42.28
PL-V20	1,048,576	79,317,960	100	75.64



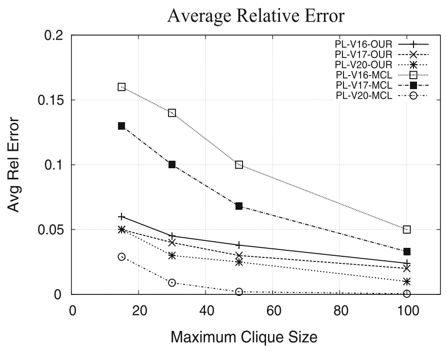


Fig. 3 Outdegree histograms (*left*) and average relative error (*right*) in synthetic graphs

clustering we use deals only with unlabeled graphs. We compare our clustering against MCL clustering,[6] by changing the first steps (finding clusters) in our discovery algorithm.

To measure how similar are discovered and real clique sets, we compute the average relative error (ARE), which is the average of the absolute difference between true and discovered cliques:

$$ARE = \frac{1}{|R|} \sum_{i \in R} \frac{|r_i - \hat{r}_i|}{r_i}, \tag{1}$$

where r_i and $\hat{r}_i$ are the real and discovered clique sizes, and $|R|$ is the number of real cliques. We consider a real clique to be "discovered" if we find more than half of its vertices.

We also evaluate the discovery algorithm based on precision and recall:

$$precision = \frac{\sum_{i \in R} |RCE \cap DCE|}{\sum_{i \in R} |DCE|}, \tag{2}$$

$$recall = \frac{\sum_{i \in R} |RCE \cap DCE|}{\sum_{i \in R} |RCE|}, \tag{3}$$

[6] Available at http://micans.org/mcl/.

Table 4 Time required per retrieved clique of different sizes

Name	MC	$\|A\|$	avg	tms	$\|A\|$M	avgM	tmsM	$ptms$M
PL-V16	15	6,501	9.00	236.1	5,810	7.96	4,359.2	1,938.5
PL-V16	30	3,766	16.53	336.4	3,596	15.18	7,877.3	3,129.1
PL-V16	50	2,389	26.58	305.1	2,331	25.40	11,190.4	5,089.2
PL-V16	100	1,261	51.08	590.0	1,242	50.80	19,839.7	9,363.1
PL-V17	15	13,071	9.00	120.5	12,032	8.30	2,048.4	977.9
PL-V17	30	7,565	16.53	129.8	7,321	15.83	3,226.3	1,612.3
PL-V17	50	4,776	26.70	203.1	4,706	26.21	4,886.3	2,394.1
PL-V17	100	2,492	51.85	318.2	2,481	51.89	10,153.5	4,446.1
PL-V20	15	104,771	9.06	103.1	103,437	9.31	580.2	103.6
PL-V20	30	60,773	16.56	150.3	60,614	16.97	614.6	152.4
PL-V20	50	38,524	26.62	155.4	38,473	27.09	639.7	248.2
PL-V20	100	20,095	51.62	178.6	20,097	52.11	1,371.1	505.7

where RCE is the node set of a real clique and DCE is the node set of the corresponding discovered clique.

In addition, we compare the number of discovered cliques ($|A|$) with respect to real cliques:

$$recallNumCliques = \frac{|A|}{|R|} . \tag{4}$$

In order to compare the clustering algorithms, we first measure execution times. We execute the version of the discovery algorithm that uses MCL only with one iteration with $I = 2.0$ (default setting for *Inflation* parameter). We also execute our clustering, where we use 40 to 100 iterations in order to reach similar clustering quality (yet, our iterations are much faster than that of MCL). Table 4 shows the number of discovered cliques ($|A|$), average sizes (avg), and the average time in milliseconds (tms) to retrieve a clique when using our dense subgraph algorithm. We also add the corresponding values obtained using MCL clustering ($|A|$M, avgM). The MCL execution time (tmsM) considers sequential time, whereas $ptms$M considers parallel execution time with 16 threads. Our current discovery algorithm implementation is sequential; its parallel version, which is under construction, should improve execution times. Still, already our sequential algorithm is an order of magnitude faster than sequential MCL. Our approach works better than MCL for graphs that have fewer cliques, as in PL-V16 and PL-V17. In such cases, even our sequential time with multiple iterations is much faster than one iteration of the parallel MCL with 16 threads. For graphs that contain more cliques and small MC values, the time of our sequential algorithm is comparable to parallel MCL using 16 threads; yet, as the cliques grow, MCL does not scale well and even its parallel version becomes slower than ours.

Figure 3 (right) shows that ARE (Eq. 1) values are very low in our strategy (<0.06, i.e., 6 %) and the error grows slightly when the number of cliques increases in graphs. However, changing our clustering algorithm to MCL, the average relative error increases when the graph contains smaller or fewer cliques hidden in the graph. On the other hand, in all cases, we have a precision of 1.0, which means that we only recover existing cliques. Figure 4 (left) shows recall (Eq. 3), and again, we observe that our discovery algorithm behaves very well (more than 0.93, i.e., 93 %) for different number and size of cliques hidden in the graphs. In contrast, MCL is very sensitive to the number and size of cliques, being less effective

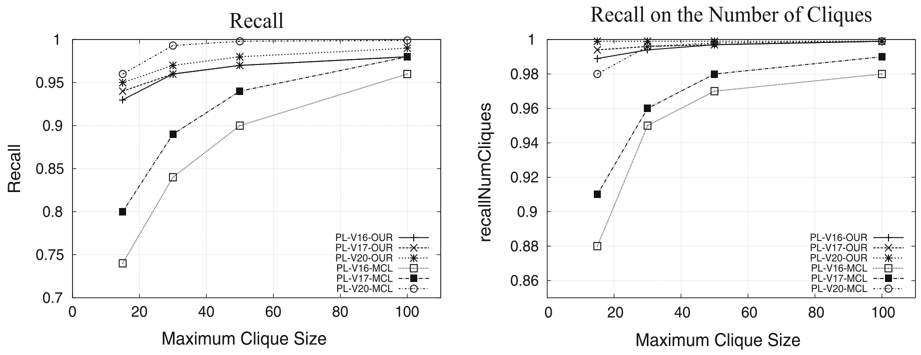


Fig. 4 Recall on the number of vertices (*left*) and on the number of cliques (*right*) discovered in synthetic graphs

Table 5 Compression (bpe) and time using MCL with different inflation I values for dblp-2011

Metric	Inflation (I)					Ours
	1.2	1.4	2.0	3.0	4.0	
bpe	8.76	9.43	10.17	10.44	10.51	8.41
tms	116,093	36,258	11,643	5,736	5,671	5,449
ptms	17,313	5,509	2,072	1,526	1,710	

Table 6 Compression (bpe) and time using MCL with different inflation values I for eu-2005

Metric	Inflation (I)					Ours
	1.2	1.4	2.0	3.0	4.0	
bpe	3.46	3.13	3.18	3.21	3.25	2.67
tms	–	–	–	–	–	2,874
ptms	65,359	62,297	59,535	59,285	89,066	–

for fewer or smaller cliques. We see a similar behavior in Fig. 4 (right), where we measure $recallNumCliques$ (Eq. 4).

To summarize, with our discovery strategy, we discover 98–99 % of the cliques [Fig. 4 (right)] and find their correct vertices with average relative errors between 1 and 6 % [Fig. 3 (right)]. The performance is better for larger cliques. One possible reason is that the clustering algorithm we use tends to find greater similarity on those adjacency lists that have more vertices in common.

We also evaluate the impact in scalability and compression (described in Sect. 5) using MCL over a real undirected social graph (dblp-2011, see Table 7). We execute MCL with different values for the inflation parameter (I). Table 5 shows the compression (bpe) and sequential execution time (tms) and parallel execution with 16 threads (ptms). It also shows that our clustering approach outperforms MCL, achieving less space than its slowest construction within the time of its fastest construction.

To confirm the scalability problems of MCL, we also execute it over a larger graph, namely eu-2005 (which is the smallest Web graph we use, see Table 7). We use different I values, from $I = 1.2$ to $I = 4.0$ (using $I = 6.0$ takes more than 2 days). We use parallel MCL with 16 threads; sequential MCL was disregarded since the parallel execution is already several orders of magnitude slower than our sequential algorithm. Table 6 shows the results, where

Table 7 Main statistics of the Web graphs we used in our experiments

Dataset	$\|V1\|$	$\|E1\|$	$d1$	$\|E2\|$	$d2$
eu-2005	862,664	19,235,140	22.30	18,733,713	21.72
indochina-2004	7,414,866	194,109,311	26.18	191,606,827	25.84
uk-2002	18,520,486	298,113,762	16.10	292,243,663	15.78
arabic-2005	22,744,080	639,999,458	28.14	631,153,669	27.75

The average neighbors per node are $d1$ and $d2$

we also give the achieved compression in *bpe* using our compressed structure with compact data structures (Sect. 5). Using the compression scheme described in Sect. 4 is an order of magnitude faster. This confirms that the clustering we use in our discovery algorithm is much more scalable than MCL.

The MCL scalability issue has been reported in several works [34,42,44,46]. In fact, Mishra et al. [46] reports that MCL performs poorly in sparse graphs. Macropol and Singh [42] proposed a scalable discovery algorithm for best clusters (based on a score metric) for labeled graphs. Their clustering algorithm is similar to ours, but for labeled graphs. They use Local Sensitive Hashing (LSH) and achieve better performance than MCL. Additionally, the time complexity of our algorithm is $O(E \log E)$, while a straightforward implementation of MCL is $O(V^3)$ time, as mentioned in the MCL web site FAQ section.[7] Another issue with MCL is that it does not guarantee good effectiveness on directed graphs.[8]

4 Using virtual nodes

In this section, we describe compact graph representations based on using virtual nodes to compress the dense subgraphs. Depending on the representation of the final graph, we obtain various structures supporting out-neighbor and out/in-neighbor navigation.

In a first phase, we apply the discovery of dense subgraphs explained in Sect. 3. Then, we apply the idea of virtual nodes [16] over the original graph, to factor out the edges of the dense subgraphs found. Given a dense subgraph $H(S, C)$, we introduce a new virtual node w in V and replace all the edges in $S \times C$ by those in $(S \times \{w\}) \cup (\{w\} \times C)$.

As the result is still a graph, we iterate on the process. On each iteration, we discover dense subgraphs in the current graph and replace their edges using virtual nodes. We refer to this approach as *dense subgraph mining* (*DSM*).

The outcome of this phase is a graph equivalent to the original one, in the sense that we must expand paths that go through virtual nodes to find all the direct neighbors of a node. The new graph has much fewer edges and a small amount of virtual nodes in addition to the original graph nodes.

On a second phase, we apply different state-of-the-art compression techniques and node orderings over this graph to achieve compression and fast out- and out/in-neighbor queries.

This process has three parameters: ES specifies the minimum size $|S| \cdot |C|$ of the dense subgraphs we want to capture during the discovery, T is the number of iterations we carry out to discover dense subgraphs, and P is the number of hashes used in the clustering stage of the dense subgraph discovery algorithm.

[7] http://micans.org/mcl/man/mclfaq.html#howfast.

[8] http://micans.org/mcl/man/mclfaq.html#goodinput.

Table 8 Main statistics on the *DSM* reduced graphs

Dataset	T	$\|V3\|$	$\|E3\|$	$d3$	$\|E2\|/\|E3\|$	$\|VN\|$	ET (min)
eu-2005	10	1,042,260	3,516,473	3.37	5.32	179,596	3.45
	5	1,019,699	3,776,194	3.70	4.96	157,035	2.45
indochina-2004	10	8,079,568	21,313,402	2.63	8.99	664,703	35.0
	5	8,030,729	22,186,260	2.76	8.63	615,864	24.3
uk-2002	10	19,842,886	54,391,059	2.74	5.37	1,322,400	65.8
	5	19,767,439	56,329,408	2.84	5.18	1,246,953	44.2
arabic-2005	10	26,193,219	74,071,714	2.82	8.52	3,449,139	185.1
	5	25,805,521	78,919,645	3.05	7.99	3,061,441	130.3

As explained, we input the graph in natural ordering to the *DSM* algorithm. If we retain this order on the output and give virtual nodes identifiers larger than those of the original nodes, we can easily distinguish which nodes are virtual and which are original. If, instead, use a different ordering on the output, such as BFS, we need an additional bitmap to mark which nodes are virtual.

4.1 Dense subgraph mining effectiveness

In the experiments of this section, we use Web graph snapshots available from the *WebGraph* project.[9] Table 7 gives the main statistics of the Web graphs used. We define $G1(V1, E1)$ as the original Web graph and $G2(V2, E2)$ as the result of removing the (u, u) edges from $G1$ (as explained, we will store a bitmap marking which of those edges were originally present). Algorithm *DSM* will operate on $G2$ (where it will start by adding (u, u) for every node). We call $G3(V3, E3)$ the outcome of the *DSM* algorithm, where $V3 = V1 \cup VN$, VN are the virtual nodes added, and $E3$ are the resulting edges in $G3$. We always use $P = 2$ for *DSM*.

Table 8 shows the main statistics of $G3$, using $ES = 6$ and carrying out T iterations. The table also shows the number of virtual nodes ($|VN|$), the resulting average arity ($d3$), the size gain estimation based on the edge reduction, given by $|E2|/|E3|$, and the total execution time (ET) in minutes. The edge reduction is significant, from 5X to 9X, whereas the increase in nodes is moderate, 7–20 %.

4.2 Performance evaluation with out-neighbor support

In this section, we evaluate the space and time performance when supporting out-neighbor queries, by applying *DSM* and then state-of-the-art compression on the resulting graph. For the second phase, we use BV (version 3.0.1 from *WebGraph*, which uses LLP ordering [9]) and AD (version 0.2.1 of their software,[10] giving it the input in natural order [4]). We compare our results with the best alternatives, including BV [9], AD [4], and GB [32]. Combining *DSM* with GB was slightly worse than GB standalone, so we omit that combination. We also omit other representations that have been superseded over time [21].

Table 9 shows the compression achieved with the combinations. The parameters for each of the techniques are tuned to provide the best performance. We refer to BV as applying BV with parameters $m = 100$ and $w = 7$, where m is the maximum reference chain and w is

[9] Available at law.dsi.unimi.it.

[10] Available at http://www.dia.uniroma3.it/~drovandi/software.php.

Table 9 Compression performance in bpe, with support for out-neighbor queries

Dataset	eu-2005	indochina-2004	uk-2002	arabic-2005
BV_{m100w7}	3.74	1.50	2.38	1.79
AD_8	3.64	1.60	2.64	2.26
GB_{128}	**1.83**	*1.09*	**1.76**	**1.35**
DSM+ESx-T10+BV	3.06	1.48	2.68	2.06
DSM-ESx-T5+AD_4	2.44	1.18	2.05	1.56
DSM-ESx-T5+AD_8	2.30	1.06	1.87	1.45
DSM-ESx-T10+AD_4	2.32	1.14	2.01	1.51
DSM-ESx-T10+AD_8	*2.20*	**1.03**	*1.83*	*1.40*

The best-performing one per graph is in bold and the second best in italics

the window size [those parameter values improve compression, but increase access times a little, as observed in Fig. 5 (left)]; AD_l as using AD with parameter l; and GB_h as using GB with parameter h. For our representations, we add a bitmap of length $|V|$ marking which nodes have a self-loop (as our technique otherwise loses this information). We use RRR for compressing the self-loop bitmap. We compute bits per edge (bpe) as the total amount of bits of the compressed graph plus the self-loop bitmap, divided by $E1$.

We refer to DSM-ESx-Ty as using $ES = x$ and iterating *DSM* for $T = y$ times. We tuned our combinations using DSM with BV_{m3w7} (DSM-ESx-Ty+BV) and DSM with AD_8 (DSM-ESx-Ty+AD_8). Using DSM with BV, we found that the best ES values were 30 for eu-2005 and 100 for indochina-2004, uk-2002, and arabic-2005; while the best T value was 10. On the other hand, the best ES value when combining DSM with AD was 10 for eu-2005 and arabic-2005, and 15 for indochina-2004 and uk-2002. Those are the x values that correspond to ESx in the table.

Table 9 shows GB outperforms BV and AD by a wide margin. Among our representations, the one using $T = 10$ combined with AD_8 gives the best results. Overall, in most datasets, the best compression ratio for accessing out-neighbors is achieved by GB_{128}, but our technique is very close for datasets *uk-2002* and *arabic-2005*, and we slightly outperform it for *indochina-2004*. Only for the smallest graph, *eu-2005* is GB_{128} better by far. Nevertheless, as observed in Fig. 5 (right), over transposed graphs, our technique achieves better compression and access time than GB_h, and the sum favors our techniques when supporting in- and out-neighbors (i.e., when storing both the direct and reverse graphs).

Figure 5 (left) shows the space/time tradeoffs achieved using BV, AD, and GB (using parameter value $h = 8, 32, 64, 128$), compared to using *DSM* before applying BV or AD. When combining DSM with BV, we used the optimum ES values mentioned above and used BV with parameters $w = 7$, and $m = 3$, 100, and 1,000. When combining with AD, we also use the optimum ES value and test different values of l for AD in the second phase. We did not use a greater T because the edge reduction obtained did not compensate the extra virtual nodes added. We compute the time per edge by measuring the total time, t, needed to extract the out-neighbors of all vertices in $G1$ in a random order, and then dividing t by the total number of recovered edges (i.e., $|E1|$).

We observe that both BV and AD improve when combined with *DSM*. In particular, the combination of *DSM* with AD dominates BV, AD, and *DSM* plus BV. It achieves almost the same space/time performance as GB, which dominates all the others, and surpasses it in graph in-2004. Only in the smallest graph, eu-2005, does GB clearly dominate our combination.

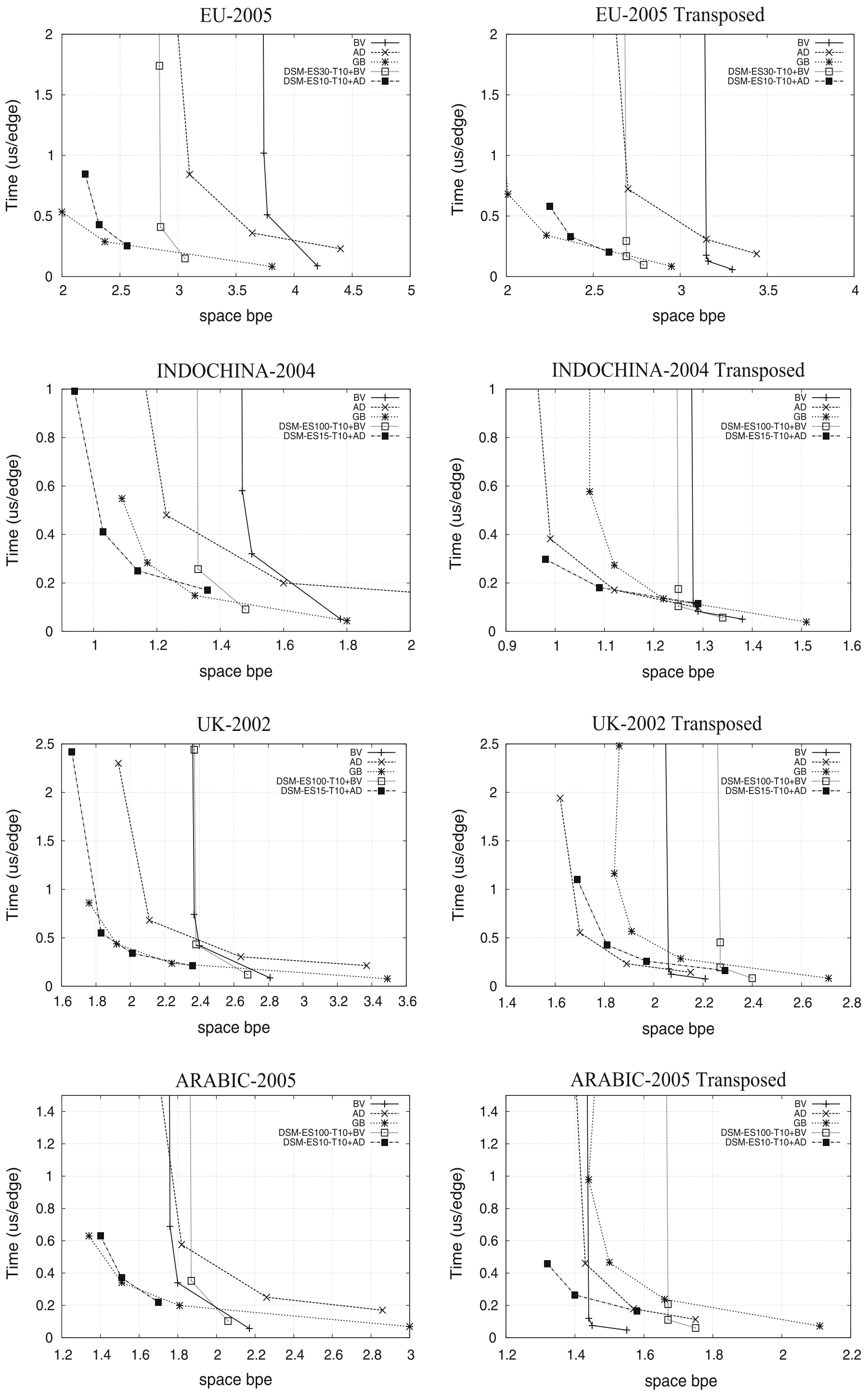


Fig. 5 Space/time efficiency with out-neighbor queries

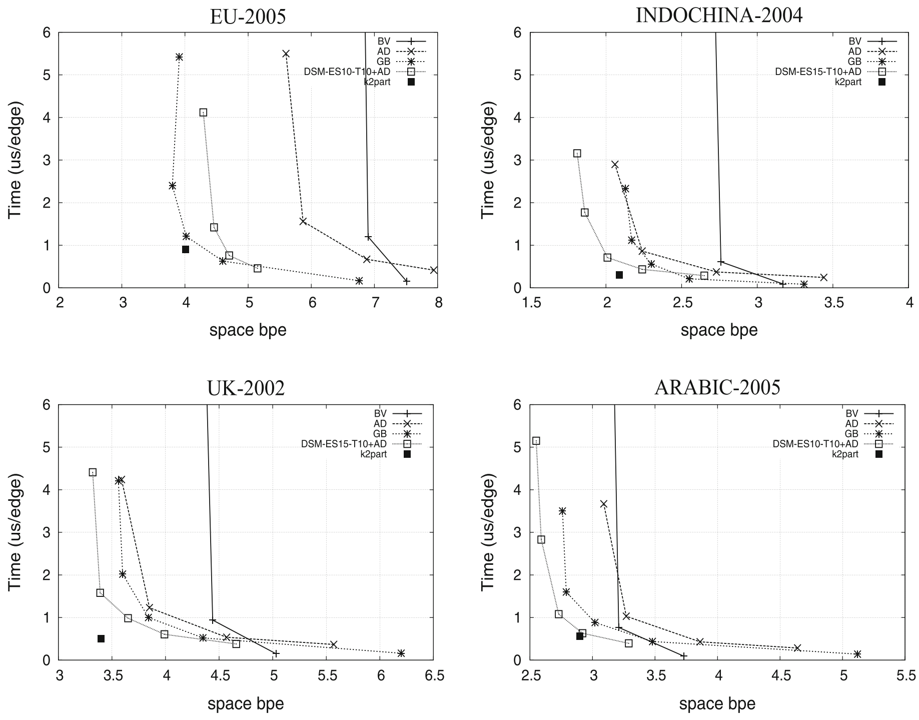


Fig. 6 Space/time efficiency with out/in-neighbor queries

Figure 5 (right) shows the same results on the transposed graphs. Note that, the *DSM* preprocessing is the same for the original and the transposed graphs, so we preprocess the graph once and then represent the reduced original and transposed graphs. On the transposed graphs, we observe that the alternative that combines DSM with BV actually performs worse than plain BV on large graphs. GB does not perform as well as on the original graphs, but on *eu-2005*, it is the best alternative. AD behaves very well on *uk-2002*, but our best combination outperforms it over the other datasets. In fact, our best combination is one of the two best alternatives in all datasets.

Figure 6 shows the space required to store the original plus the transposed graphs, combined with the time for out-neighbor queries (which is very similar to that for in-neighbor queries; these are run on the transposed graph). It can be seen that our new combinations of *DSM* plus AD dominate most of the space/time tradeoff, except on eu-2005. However, a data structure specific for out/in-neighbor queries (*k2part* [23]) offers comparable (and in some graphs much better) time performance, but we outperform it in space, considerably on some graphs.

Next, we will consider a truly bidirectional representation for the reduced graph, obtaining much less space with higher query time.

4.3 Performance evaluation with out/in-neighbor support

In this section, we combine the output of *DSM* with a compression technique that supports out/in-neighbor queries: the *k2tree* [11]. We use the best current implementation [12]. We apply dense subgraph discovery with parameters $ES = 10, 15, 100$ and $T = 5, 10$. In all

Table 10 Compression performance when combining with *k2trees*

Dataset	eu-2005	indochina-2004	uk-2002	arabic-2005
k2treeNAT	3.45	1.35	2.77	2.47
k2treeBFS	3.22	1.23	2.04	1.67
DSM-ES10-T5 + k2treeNAT	2.76	1.36	2.40	1.76
DSM-ES10-T10 + k2treeNAT	2.71	1.34	2.40	1.76
DSM-ES15-T5 + k2treeNAT	2.65	1.27	2.28	1.67
DSM-ES15-T10 + k2treeNAT	2.59	1.27	2.27	1.66
DSM-ES100-T5 + k2treeNAT	2.56	1.16	2.13	1.52
DSM-ES100-T10 + k2treeNAT	2.48	1.14	2.08	1.47
DSM-ES10-T5 + k2treeBFS	2.21	0.90	1.56	1.12
DSM-ES10-T10 + k2treeBFS	**2.11**	**0.87**	**1.53**	**1.08**
DSM-ES15-T5 + k2treeBFS	**2.11**	**0.87**	1.54	1.14
DSM-ES15-T10 + k2treeBFS	2.21	0.89	1.57	**1.08**
DSM-ES100-T5 + k2treeBFS	2.54	0.95	1.67	1.21
DSM-ES100-T10 + k2treeBFS	2.45	0.93	1.64	1.18

cases, process *DSM* is run over the graph in natural order. We denote *k2treeBFS* the variant that switches to BFS order on $G3$ when applying the *k2tree* representation, and *k2treeNAT* the variant that retains natural order.

Table 10 shows the compression achieved. We observe that the compression ratio is markedly better when using BFS ordering. In particular, the setting $ES = 10, T = 10$, and *k2treeBFS* is always the best. The space is also much better than that achieved by representing the original plus transposed graphs in Sect. 4.2.

Figure 7 shows the space/time tradeoff when solving out-neighbor queries (in-neighbor times are very similar). We include *k2treeNAT* [11], *k2treeBFS* [12], *k2part* [23], and disregard other structures that have been superseded by the last *k2tree* improvements [20]. We also include in the plots one choice DSM-ESx-Ty+AD from Sect. 4.2, which represents the direct and transposed graphs using *DSM* and $T = 10$ combined with AD using various values of l.

All those structures are clearly superseded in space by our new combinations of *DSM* and *k2treeBFS* or *k2treeNAT*. Again, the combination with BFS gives much better results, and using different ES values yields various space/time tradeoffs. On the other hand, these smaller representations reaching 0.9–1.6 bpe on the larger graphs are also significantly slower, requiring 5–20 μs per retrieved neighbor.

4.4 Scalability

Even if we aim at fitting the final compressed graph in main memory, the original graph $G2$ may be much larger and prevent a direct in-memory application of the first phase of the algorithm, *DSM*. We consider this problem in this section.

A simple approach to this problem is to maintain $G(V, E) = G2(V2, E2)$ on disk and use the main memory to keep the matrix of hash values of size $P \times |V|$ described in *Step 1* (recall Sect. 3.2), taking advantage of the fact that $|V| \ll |E|$. Given that each row of the $P \times |V|$ matrix (formed by P hashes associated with an adjacency list) can be computed independently of each other, and this step requires only one traversal over the graph. This step is also suitable for data streaming or for computing each group of rows in parallel.

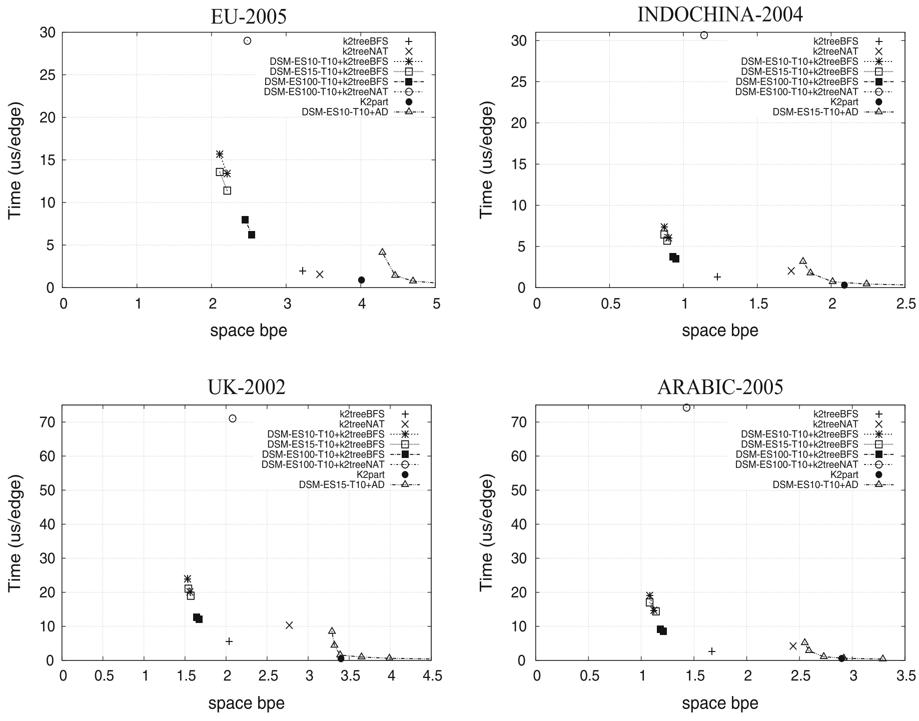


Fig. 7 Space/time efficiency with out/in-neighbor queries

Step 2 runs on main memory for storing and sorting the matrix by columns. Once the matrix has been sorted, we proceed to create the actual clusters in *Step 3*, where we need to access the actual graph stored on disk. Thus, after *Step 2*, we obtain the set of node ids for each cluster. With this information, we can load from disk only the blocks we need for a set of clusters. In this part, it is important that, thanks to the locality of reference found on Web graphs, there is a high probability that clusters are formed by nearby adjacency lists that reside on the same or a few disk blocks. We refer to this number of disk blocks as k. *Steps 3* and *4* require to keep blocks where current clusters reside in order to find dense subgraphs and replace adjacency lists with virtual nodes and their definitions. Since replacing with virtual nodes reduces edges, the graph is smaller at the end of each iteration. After the replacements are done, disk blocks are written back to disk. Thus, considering T iterations and k disk blocks for maintaining adjacency lists, the worst-case I/O cost of the complete algorithm is $O(T((|E| + |V|)/B + k))$, where B is the disk block size. The algorithm needs only a few iterations in practice (at most $T = 10$), and k is usually rather small, which makes the algorithm almost I/O optimal in practice.

However, since Web graphs expose locality of reference, we can also divide the graph into multiple parts and process each part independently, at the cost of losing some inter-part dense subgraphs. Doing so, we can reduce the memory and processing time according to the needs of each part. Processing each part independently is also attractive for parallel and distributed processing.

This is done in three stages. First, we apply *DSM* (in main memory or on disk) over each part (parts can be just node ranges in natural order). Second, we remap virtual node

Table 11 Compression of graph eu-2005 divided in different number of parts

NP	$\max(\lvert V2\rvert + \lvert E2\rvert)$	k	$\lvert V3\rvert + \lvert E3\rvert$	$\lvert E2\rvert/\lvert E3\rvert$	$\lvert VN\rvert$	bpe
1	19,514,936	105,653	4,620,439	5.32	179,596	2.20
5	5,057,710	50,971	4,687,354	5.10	155,944	2.25
10	3,561,390	32,290	4,709,537	5.07	154,821	2.28
20	2,674,977	19,308	4,783,414	4.96	148,615	2.29

identifiers so that they are globally unique. Third, we merge all the reduced graphs and apply AD reordering and encoding.

We evaluated the partitioning scheme to measure the impact of locality of reference on how well compression and disk block requirements behave. In this case, we took the smallest Web graph, eu-2005, and evaluated compression using different numbers of parts. We separate the nodes dividing the node identifiers by the number of parts, NP. We first apply DSM-ES15-T10 (with $ES = 15$ and $T = 10$) on all parts, then remap the nodes, and finally merge and apply AD$_8$. Table 11 shows the number of disk blocks k (for a block size of 4 KB) required for sets of 1,000 clusters. The value of k displayed in Table 11 considers the first iteration and all parts. It shows that, when we use 20 parts, we can still obtain good results on reducing edges, disk block requirements, and compression performance measured in bpe. Since our last stage, using AD, is applied over the merged edge-reduced graph, the memory requirement depends basically on the edge compression gain ($|E2|/|E3|$). We also show the space requirement for the input graph as $\max(|V2|+|E2|)$ on a part and the number of nodes and edges required to store G_3 ($|V3| + |E3|$).

We also experimented with a larger dataset, uk-2005-05,[11] which has 77,741,046 nodes and 2,965,197,340 edges. We divide the graph into 10 parts. This yields parts with a minimum of about 217 and a maximum of about 410 million edges. We achieve 1.65 bpe and a neighbor retrieval time of about 0.54 µs. These results show that using, say, DSM − ES15 − T10 plus AD_8 provides a scalable approach for large Web graphs. In contrast, using AD_8 standalone, we obtain 2.34 bpe. Using BV standalone, we achieve 2.12 bpe at maximum compression, where queries are not supported. Using GB with $h = 64$, we achieve 1.75 bpe and a neighbor retrieval time of 0.36 µs, whereas using $h = 128$ the bpe is 1.59 and query time is 0.65 µs. Therefore, the main conclusions we had reached, that our new scheme and GB provide similar performance on Web graphs and dominate all the other approaches, seem to be robust and remain valid on much larger graphs.

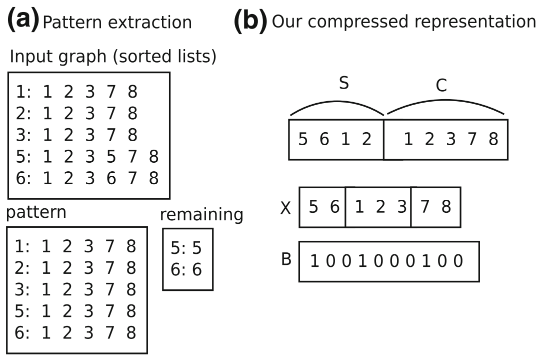
The conclusions obtained on bidirectional representations also remain valid, that is, our representations supporting out/in-neighbor queries are much smaller yet slower. Combining the results of *DSM* with 10 parts with *k2treeBFS* on graph uk-2006-05 yields 1.29 bpe and a neighbor retrieval time of 12.4 µs. The standalone *k2treeBFS* obtains 1.78 bpe with a retrieval time of 4.12 µs.

5 Compact data structure for dense subgraphs

In this section, we present a new compressed graph representation based on dense subgraphs that supports out/in-neighbor as well as various mining queries. We extract dense subgraphs essentially as in Sect. 3 and represent them using compact data structures based on bitmaps

[11] Available at http://law.dsi.unimi.it/webdata/uk-2006-05.

Fig. 8 Dense subgraph representation



and symbol sequences (described in Sect. 2.2). Recalling Definition 3.1, 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. This representation will not use virtual nodes, and its output is not anymore a graph. As a result, we cannot iterate on the discovery algorithm in order to find dense subgraphs involving virtual nodes.

5.1 Extracting dense subgraphs

We extract dense subgraphs using the algorithms described in Sect. 3. We use three parameters: P, the number of hashes in the clustering stage of the dense subgraph discovery, a list of ES values, where ES is the minimum $|S|\cdot|C|$ size of dense subgraphs found, and $threshold$. Parameters P and ES are the same as before; yet, now we use a decreasing list of ES values. The discovery algorithm continues extracting subgraphs of a size ES_i until the number of subgraphs drops below $threshold$ on a single iteration; then, ES is set to the next value in the list for the next iteration. Note that, in this case, we do not use the parameter T (number of iterations), since the number of iterations will depend on the number of extracted subgraphs on each iteration and the $threshold$ value. The goal of having the ES list in decreasing order is to avoid that extracting a small dense subgraph precludes the identification of a larger dense subgraph, which gives a higher benefit. Note that, this was not so critical in Sect. 4, where we were allowed to iterate over the dense subgraph discovery process and let virtual nodes participate in larger dense subgraphs.

5.2 Representing the graph

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 5.1 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.

Figure 8a shows the adjacency list representation for the graph presented in Fig. 1, where we have already added the self-loops. We also show a dense subgraph, and a remaining subgraph. Figure 8b shows our compact representation.

Algorithm 1: Construction of X and B

```
Input: Subsets $S_1, \ldots, S_N$ and $C_1, \ldots, C_N$
Output: Sequence $X$ and Bitmap $B$
$X \leftarrow \varepsilon$;
$B \leftarrow \varepsilon$;
for $i \leftarrow 0$ to $N$ do
    $L \leftarrow S_i - C_i$;
    $M \leftarrow S_i \cap C_i$;
    $R \leftarrow C_i - S_i$;
    $X \leftarrow X : L : M : R$;
    $B \leftarrow B : 10^{|L|}10^{|M|}10^{|R|}$ ;
end
return $X, B$;
```

5.3 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 5.1. 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 separations in 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. Figure 8b shows this representation for our example, which has just one dense subgraph. Algorithm 1 describes how X and B are built.

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 bidirectional compressed graph representation.

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 [33] for sequence X and compressed bitmap representation RRR [49] for bitmap B. The total space is $|X|H_0(X) + o(|X|\log\sigma) + |X|H_0(B)$ bits, 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 can be much less than the number of edges in the subgraph it represents.

5.4 Neighbor queries

We answer out/in-neighbor queries as described by Algorithms 2 and 3. 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.

We explain how the out-neighbors algorithm works; the case of in-neighbors is analogous. Using $select_X(u, i)$, we find all the places where node u is mentioned in X. This corresponds

Algorithm 2: Find out-neighbors

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

Algorithm 3: Find in-neighbors

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

Algorithm 4: Get cliques and bicliques

```
Input: Sequence $X$, bitmap $B$ and vertex $u$
Output: List of $allcliques$ and $allbicliques$
$allcliques \leftarrow \langle\rangle$;
$allbicliques \leftarrow \langle\rangle$;
$n \leftarrow rank_B(1, |B|)$;
$cur \leftarrow 1, p1 \leftarrow 0$;
while $cur < n$ do
    $p2 \leftarrow select_B(1, cur + 1)$;
    $p3 \leftarrow select_B(1, cur + 2)$;
    $p4 \leftarrow select_B(1, cur + 3)$;
    if $p2 - p1 = 1 \wedge p4 - p3 = 1$ then
        $s \leftarrow p2 - (cur + 1) + 1$;
        $e \leftarrow p3 - (cur + 2)$;
        $clique \leftarrow \emptyset$;
        for $i \leftarrow s$ to $e$ do
            $clique \leftarrow clique \cup \{access_X(i)\}$;
        end
        $allcliques.add(clique)$;
    end
    else if $p3 - p2 = 1$ then
        $s \leftarrow p1 - cur + 1$;
        $m \leftarrow p2 - (cur + 1)$;
        $e \leftarrow p4 - (cur + 3)$;
        $biclique.S \leftarrow \emptyset, biclique.C \leftarrow \emptyset$;
        for $i \leftarrow s$ to $m$ do
            $biclique.S \leftarrow biclique.S \cup \{access_X(i)\}$;
        end
        for $i \leftarrow m + 1$ to $e$ do
            $biclique.C \leftarrow biclique.C \cup \{access_X(i)\}$;
        end
        $allbicliques.add(biclique)$;
    end
    else
        other type of dense subgraph ;
    end
    $cur \leftarrow cur + 3, p1 \leftarrow p4$;
end
return $allcliques, allbicliques$
```

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 in the case of out-neighbor queries.

5.5 Supporting mining queries

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. Algorithm 4 shows how easy it is to recover all cliques and bicliques stored in the compressed structure. This information can be useful for mining and analyzing Web and social graphs. The time complexity is $O(|output| \cdot \log \sigma)$.

Note that, we only report, in this simplified algorithm, pure cliques and bicliques. A slight modification would make the algorithm extract the clique $S \cap C$ that is inside dense subgraph $H(S, C)$ or the bicliques $(S - C, C)$ or $(S, C - S)$.

Another interesting query could be computing the density of the dense subgraphs stored in H. Let us use a definition of density [2] that considers the connections inside a subgraph: A subgraph $G'(V', E')$ is γ-*dense* if $\frac{|E'|}{|V'|(|V'|-1)/2} \geq \gamma$. The density of a clique is always 2.

Algorithm 5: Get all dense subgraphs with density at least γ

```
Input: Sequence $X$, bitmap $B$ and density $\gamma$
Output: List $ls$ of dense subgraphs with density at least $\gamma$
$ls \leftarrow \langle\rangle$;
$n \leftarrow rank_B(1, |B|)$;
$cur \leftarrow 1$, $p1 \leftarrow 0$;
while $cur < n$ do
    $p2 \leftarrow select_B(1, cur + 1)$;
    $p3 \leftarrow select_B(1, cur + 2)$;
    $p4 \leftarrow select_B(1, cur + 3)$;
    $V \leftarrow p4 - p1 - 3$;
    $E \leftarrow (p3 - p1 - 2) \cdot (p4 - p2 - 2)$;
    $g \leftarrow E/(V \cdot (V - 1)/2)$;
    if $(g \geq \gamma)$ then
        $ls.add((cur + 2)/3)$;
    end
    $cur \leftarrow cur + 3$, $p1 \leftarrow p4$;
end
return $ls$
```

The density of a biclique (S, C) is $\frac{2 \cdot |S| \cdot |C|}{(|S|+|C|)(|S|+|C|-1)}$. Algorithm 5 computes the density of all dense subgraphs and reports all dense subgraphs with a density over a given γ.

Some of other possible mining queries are the following:

- Get the number of cliques where node u participates. We just count the number of times node u is in the M component of X. The algorithm is similar to, say, Algorithm 2; yet, it just identifies the component where u is and increments a counter whenever this component is M.
- Get the number of bicliques where node u participates. This is basically the same as the previous query; yet, this time we count when node u is in components L or R. If u is in L, it is a *source* and if it is in R, it is a *center*.
- Get the number of subgraphs. We just compute the number of 1s in B and divide this number by 3. This is because for every dense subgraph in X, there are 3 1s in B, as shown in Fig. 8.

5.6 Dense subgraph mining effectiveness

We experiment with the same Web graphs of Sect. 4.1, plus various social networks that are also available in the *WebGraph* site. In addition, we use the LiveJournal directed graph, available from the *Stanford Network Analysis Package (SNAP)* project[12] (LiveJournal-SNAP). Table 12 lists their main statistics.

We used our dense subgraph discovery algorithm with parameters $ES = 500, 100, 50, 30, 15, 6$, discovering larger to smaller dense subgraphs. We used *threshold* $= 10$ for eu-2005, enron, and dblp-2011; *threshold* $= 100$ for indochina-2004, uk-2002, LiveJournal-2008, and LiveJournal-SNAP and *threshold* $= 500$ for arabic-2005.

Table 12 also gives some performance figures on our dense subgraph mining algorithm. On Web graphs (where we give the input to the mining algorithm in natural order), 91–95 % of the edges are captured in dense subgraphs, which would have been only slightly less if we had captured only bicliques [16]. Finding dense subgraphs, however, captures the structure of social networks much better than just finding bicliques, improving the percentage of edges captured from 46–55 to 48–65 %. Note also that the fraction of edges in dense subgraphs is

[12] Available at snap.stanford.edu/data.

Table 12 Number nodes and edges of graphs, and performance of subgraph mining algorithms

Data set	Nodes	Edges	$\lvert\mathcal{H}\rvert/\lvert E\rvert$ (bicliques) (%)	$\lvert\mathcal{H}\rvert/\lvert E\rvert$ (dense) (%)
eu-2005	862,664	19,235,140	91.30	91.86
indochina-2004	7,414,866	194,109,311	93.29	94.51
uk-2002	18,520,486	298,113,762	90.80	91.41
arabic-2005	22,744,080	639,999,458	94.16	94.61
enron	69,244	276,143	46.28	48.47
dblp-2011	986,324	6,707,236	49.88	65.51
LiveJournal-SNAP	4,847,571	68,993,773	53.77	56.37
LiveJournal-2008	5,363,260	79,023,142	54.17	56.51

On the top we list the Web graphs and at the bottom the social networks

Table 13 Fraction and average size of cliques, bicliques, and the rest of dense graphs found

Data set	Cliques		Bicliques		Dense subgraphs	
	Fraction (%)	Size	Fraction (%)	Size	Fraction (%)	Size
eu-2005	7.19	7.44	46.67	18.67	46.14	20.73
indochina-2004	6.53	5.18	34.55	22.47	58.92	20.54
uk-2002	3.56	4.47	42.16	17.84	54.28	21.92
arabic-2005	3.76	4.32	42.09	23.05	54.15	22.44
enron	0.07	3.33	67.20	13.09	32.73	20.75
dblp-2011	18.22	3.95	27.76	8.37	54.02	6.91
LiveJournal-SNAP	2.41	3.47	57.99	9.64	39.60	10.53
LiveJournal-2008	2.37	3.44	59.77	9.75	37.86	10.47

much lower on social networks, which anticipates the well-known fact that Web graphs are more compressible than social networks.

Table 13 complements this information with the fraction of cliques, bicliques, and other dense subgraphs, with respect to the total number of dense subgraphs found, as well as their average size. This shows that pure cliques are not very significant and that more than half of the times the algorithm is able to extend a biclique to a more general dense subgraph, thereby improving the space usage.

The next experiments consider the final size of our representation. For the component $\mathcal{H}$, we represent sequence X using WT or GMR, and for bitmap B, we use RG or RRR. These implementations are obtained from the library *libcds*.[13] For WT, we used the variant "without pointers." For the component $\mathcal{R}$, we use either *k2tree* [12] or MP$_k$ [23], the improvement over the proposal of Maserrat and Pei [43]. Although we use the most recent version of the *k2tree*, we use it with natural node ordering to maintain consistency between the node names in $\mathcal{H}$ and $\mathcal{R}$. An alternative would have been to use BFS ordering for both, that is, reordering before applying the dense subgraph mining, but this turned out to be less effective.

Table 14 shows how the compression evolves depending on parameter ES, on graph dblp-2011. ES values in Tables 14 and 15 represent the last value we consider in the ES list. For

[13] Available at http://libcds.recoded.cl.

Table 14 Evolution of compression as ES decreases, for the dblp-2011 data set

	ES				
	500	100	50	30	15
$\lvert X\rvert$	6.6 K	75.8 K	232.6 K	456.8 K	1.05 M
$\lvert\mathcal{H}\rvert$ in bytes	47.4 K	168.0 K	487.9 K	950.9 K	2.20 M
RE	165.8 K	636.0 K	1.24 M	1.92 M	3.25 M
$\lvert\mathcal{R}\rvert$ in bytes	7.05 M	6.88 M	6.70 M	6.50 M	6.00 M
RE/$\lvert X\rvert$	25.12	8.38	5.33	4.20	3.09
bpe	8.47	8.41	8.58	8.89	9.79

Table 15 Compression performance for Web graphs, compared to other techniques

Data set	$G = \mathcal{H} \cup \mathcal{R}$			k2treeBFS	DSM
	ES	$RE/$|X|	bpe	bpe	bpe
eu-2005	6	7.29	2.67	3.22	**2.11**
indochina-2004	6	14.17	1.49	1.23	**0.87**
uk-2002	6	8.50	2.52	2.04	**1.53**
arabic-2005	6	11.56	1.85	1.67	**1.08**

DSM refers to DSM-ES10-T10+k2treeBFS

instance, $ES = 100$, in Table 14, means that we use the sequence of values $ES = 500$, 100. As ES decreases, we capture more dense subgraphs; yet, they are of lower quality, and thus, their space saving decreases. To illustrate this, we show the length $|X| = \sum_r |S_r| + |C_r| - |S_r \cap C_r|$, the number of bytes used to represent X and B ("$|\mathcal{H}|$ in bytes", using WT for X and RRR for B), and the total edges represented by $\mathcal{H}$ (RE $= \sum_r |S_r| \cdot |C_r|$). All these indicators grow as ES decreases. Then, we show the size of $\mathcal{R}$ in bytes (using representation MP$_k$, with the best k for $\mathcal{R}$), which decreases with ES. As explained, what also decreases is RE/$|X|$, which indicates the average number of edges represented by each node we write in X. Finally, we write the overall compression performance achieved in bpe, computed as $bpe = (bits(\mathcal{H}) + bits(\mathcal{R}))/|E|$. It turns out that there is an optimum ES value for each graph, which we use to maximize compression.

Tables 15 and 16 compare the compression as we achieve with the alternatives we have chosen for Web and social graphs. We show the last ES value used for discovering dense subgraphs, the ratio RE/$|X|$, and the compression performance in bpe obtained on the Web and social graphs. We use WT and RRR where the sampling parameter is 64 for compressing $\mathcal{H}$. For compressing $\mathcal{R}$, we use *k2treeNAT* for Web graphs and MP$_k$ for social networks, which gave the best results (with enron as an exception, where using *k2treeNAT* on $\mathcal{R}$ provides better compression than MP$_k$, as displayed).

We compare the results with standalone *k2treeBFS* on Web graphs, *k2treeNAT* on enron, and MP$_k$ on the other social networks.

Our technique does not obtain space gains on Web graphs compared to *k2treesBFS*. Moreover, the variant DSM-ES10-T10+k2treeBFS of Sect. 4.3, also included in the table, is even better.

Table 16 Compression performance for social networks, compared to other techniques

Data set	$G = \mathcal{H} \cup \mathcal{R}$			MP_k	k2treeNAT	BV
	ES	RE/$|X|$	bpe	bpe	bpe	bpe
enron (with k2treeNAT)	6	2.06	**10.07**	17.02	10.31	18.30
enron	6	2.06	15.42	17.02	10.31	18.30
dblp-2011	100	8.38	**8.41**	8.48	9.83	10.13
LiveJournal-SNAP	500	12.66	**13.02**	13.25	17.35	23.16
LiveJournal-2008	100	4.88	**13.04**	13.35	13.63	17.84

BV refers to BV adapted to support out/in-neighbor queries

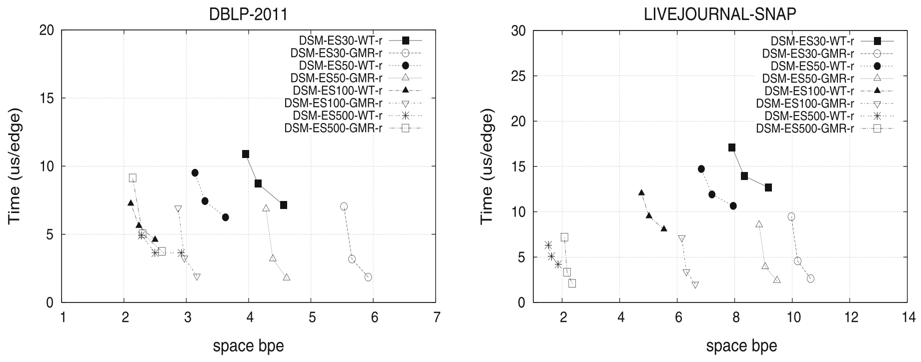


Fig. 9 Space/time efficiency with out-neighbor queries on social networks, for various ES values (only component $\mathcal{H}$ is considered)

On social networks, the gains of our new technique are more modest with respect to MP_k. However, we show next that our structure is faster too. Moreover, there are no other competing techniques as on Web graphs. Our development of Sect. 4.3 does not work at all (it reduces less than 1.5 % of edges, while increasing nodes when introducing virtual ones). The next best result is obtained with BV (which is more effective than GB and AD for social networks).

We note that BV is unable to retrieve in-neighbors. To carry out a fair comparison, we follow BV authors suggestion [9] for supporting out/in-neighbor queries. They suggest to compute the set E_{sym} of all symmetric edges, that is, those for which both (u, v) and (v, u) exist. Then, they consider the graph $G_{\mathrm{sym}} = (V, E_{sym})$ and $G_d(V, E - E_{sym})$, so that storing G_{sym}, G_d, and the transpose of G_d enables both types of queries. The space we report in Table 16 for BV considers this arrangement and, as anticipated, is not competitive.

5.7 Space/time performance

Figure 9 shows the space/time tradeoffs achieved on dblp-2011 and LiveJournal-SNAP graphs considering only the $\mathcal{H}$ component. We test different ES parameters. We use WT and GMR for the structures that represent X and RRR for B. These are indicated in the plots as WT-r and GMR-r. The sampling parameter for RRR is 16, 32, and 64, which yields a line for each combination. Along this section, we measure out-neighbor query times, as in-neighbor

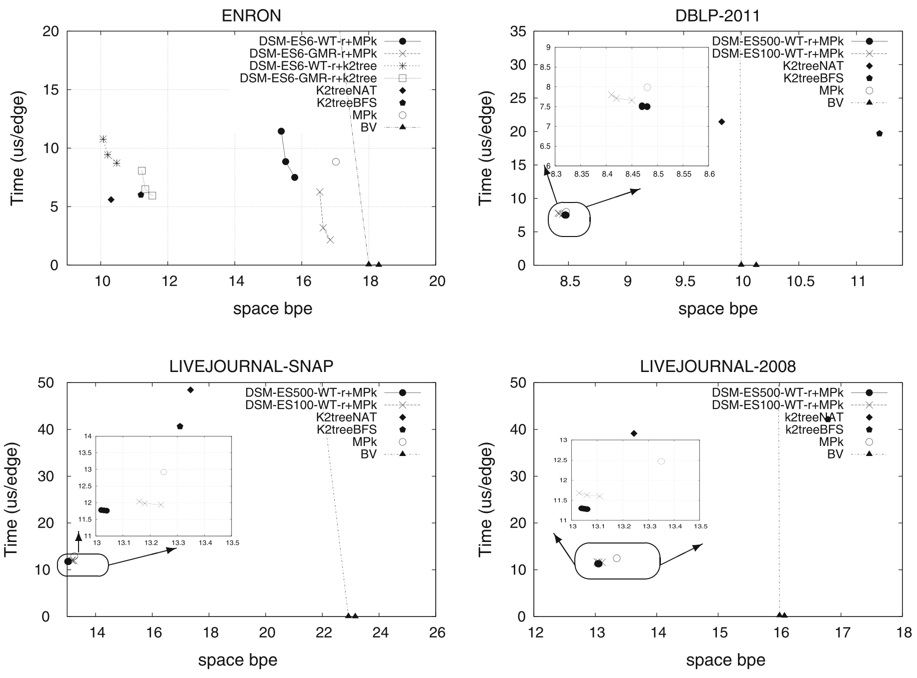


Fig. 10 Space/time tradeoffs for social networks

queries perform almost identically. We observe that using WT provides more compression than GMR, but it requires more time.

The plots show how using increasing ES improves space and time simultaneously, until reaching the optimum space. Using a larger ES value also implies fewer iterations on the dense subgraph extraction algorithm, which dominates construction time (this is currently 0.1–0.2 ms per extracted edge, but construction is not yet optimized).

We now consider our technique on social networks, representing $\mathcal{H}$ and $\mathcal{R}$, the latter using either *k2tree* or MP_k, and compare it considering space and time with the state of the art. This includes standalone *k2trees* with BFS and natural order, MP_k with the best k and, as a control value, BV with out/in-neighbor support. Now, our time is the sum of the time spent on $\mathcal{H}$ and on $\mathcal{R}$. We represent $\mathcal{H}$ using our best alternatives based on DSM-ESx-WT-r and DSM-ESx-GMR-r.

Figure 10 compares the results on social networks. The inner figures show a closeup of the best alternatives. While, on enron, *k2tree* with natural order is the best choice when using little space, on the other networks, our combination of *DSM* and MP_k is the best, slightly superseding standalone MP_k in both space and time.

Figures 11 and 12 carry out a similar study on Web graphs. In Fig. 11, we also show that on these graphs, *DSM* improves significantly in space with respect to detecting only bicliques ("BI"), while the time is similar. Figure 12 shows that the structure proposed in this section is dominated in space and time by that proposed in Sect. 4. Yet, we remind that the structure we propose in this section is able to answer various mining queries related to the dense subgraphs found easily and using no extra space.

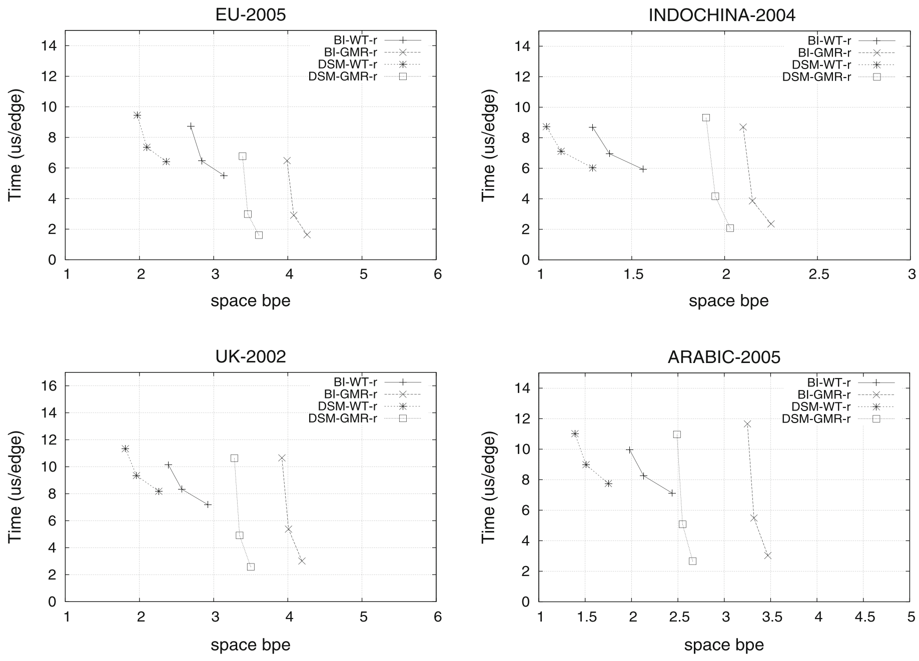


Fig. 11 Space/time efficiency with out-neighbor queries on Web graphs, for various sequence representations (only component $\mathcal{H}$ is considered)

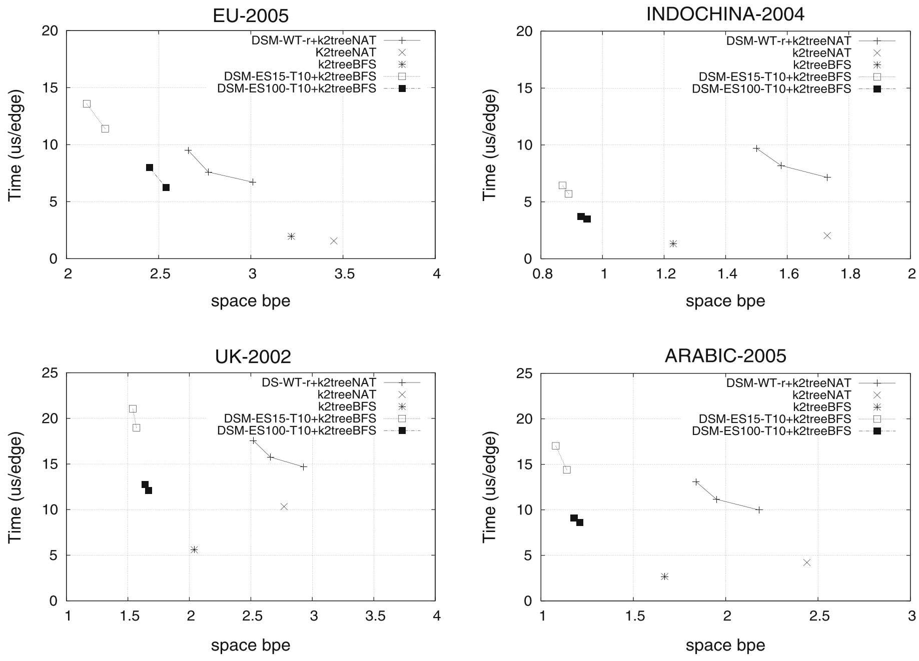


Fig. 12 Space/time tradeoffs for Web graphs

6 Conclusion

This paper studies graph compression schemes based on finding dense subgraphs. Dense subgraphs generalize the bicliques considered in previous work [16], and our experiments show that this generalization pays off in terms of compression performance. We show how previous biclique discovery algorithms can be adapted to detect dense subgraphs.

We first present a compression scheme based on factoring out the edges of dense subgraphs using virtual nodes, which turns out to be suitable for Web graphs. After iteratively reducing the graph via virtual nodes, we list the nodes in BFS order and using an encoding related to it [4]. The resulting space and time performance are very similar to the best current representation supporting out-neighbor queries (Grabowski and Bieniecki 2011). When supporting both out- and in-neighbor queries, instead, our technique generally offers the best time when using little space. In case graphs do not fit in main memory, we propose a disk-friendly approach that exploits locality of reference and data partitioning to build the compressed structure keeping almost the same compression performance. Dividing the data is also attractive for parallel and distributed processing.

If, instead, we combine the result of dense subgraph mining with a bidirectional representation, the *k2tree* [11], using BFS node ordering, the result is the most space-efficient representation of Web graphs that supports out/in-neighbors in a few microseconds per retrieved value.

We present a second compression scheme also based on dense subgraphs, yet using compact data structures instead of virtual nodes to represent them. The result turns out to be more suitable to compress social networks with out/in-neighbor support, achieving the least space while supporting queries in a few microseconds. As extracting dense subgraphs is nontrivial, and the dense subgraphs expose community substructures in social networks, these dense subgraphs may be useful for other graph mining and analysis purposes. A distinguishing feature of our representation is that it gives easy access to these dense subgraphs without any additional space.

Despite the enormous progress made in the last decade on Web graph compression, the amount of activity in this area shows that further compression is perfectly possible. The case of social networks is more intriguing, as the techniques that had been successful on Web graphs have much less impact and the best results are achieved using other properties [9,43], but still the results are much poorer. Perhaps social networks are intrinsically less compressible than Web graphs, or perhaps we have not yet found the right properties that permit compressing them further. We believe that our extension for finding more general dense subgraphs (not just bicliques) is an interesting step toward that goal. Another line of development we have contributed to is that of supporting more complex operations on the compressed representations, not only direct navigation (out-neighbors) but also bidirectional navigation and other more complex queries (such as the mining queries we support on the dense subgraphs found).

Acknowledgments Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, and Fondecyt Grant 1-110066, Chile.

References

1. Adler M, Mitzenmacher M (2001) Towards compressing web graphs. In: Proceedings of the data compression conference (DCC). Snowbird, UT, pp 203–212

2. Aggarwal C, Wang H (2010) Managing and mining graph data. Springer, Berlin
3. Anh V, Moffat A (2010) Local modeling for webgraph compression. In: Proceedings of the data compression conference (DCC). Snowbird UT, p 519
4. Apostolico A, Drovandi G (2009) Graph compression by BFS. Algorithms 2(3):1031–1044
5. Bader D, Madduri K (2005) Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In: Proceedings of the 12th international high performance computing (HiPC). Goa, India, pp 465–476
6. Becchetti L, Castillo C, Donato D, Baeza-Yates R, Leonardi S (2008) Link analysis for web spam detection. ACM Trans Web 2(1):2
7. Boldi P, Vigna S (2004) The Webgraph framework I: compression techniques. In: Proceedings of the 13th international conference on the world wide web (WWW), New York, NY, pp 595–602
8. Boldi P, Santini M, Vigna S (2009) Permuting web graph. In: The 6th workshop on algorithms and models for the web graph (WAW), Barcelona, Spain, pp 116–126
9. Boldi P, Rosa M, Santini M, Vigna S (2011) Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In: Proceedings of the 20th international conference on world wide web (WWW), Hyderabad, India, pp 587–596
10. Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. Comput Netw 1(7):107–117
11. Brisaboa N, Ladra S, Navarro G (2009) K2-trees for compact web graph representation. In: Proceedings of the 16th international symposium on string processing and information retrieval (SPIRE), Saariselkä, Finland, pp 18–30
12. Brisaboa N, Ladra S, Navarro G (2012) Personal communication including code
13. Broder A (2000) Min-wise independent permutations: theory and practice. In: Proceedings of the 27th international colloquium on automata, languages and programming (ICALP), Geneva, Italy, p 808
14. Brohée S, Van Helden J (2006) Evaluation of clustering algorithms for protein-protein interaction networks. BMC Bioinformatics 7:488
15. Bron C, Kerbosch J (1973) Finding all cliques of an undirected graph (Algorithm 457). Commun ACM 16(9):575–576
16. Buehrer G, Chellapilla K (2008) A scalable pattern mining approach to web graph compression with communities. In: Proceedings of the international conference on web search and web data mining (WSDM), Palo Alto, CA, pp 95–106
17. Cha M, Mislove A, Gummadi P (2009) A measurement-driven analysis of information propagation in the Flickr social networking. In: Proceedings of the 20th international conference on world wide web (WWW), Madrid, Spain, pp 721–730
18. Chakrabarti D, Zhan Y, Faloutsos C (2004) R-MAT: a recursive model for graph mining. In: Proceedings of the 4th SIAM international conference on data mining (SDM), Lake Buena Vista, FL
19. Chierichetti F, Kumar R, Lattanzi S, Mitzenmacher M, Panconesi A, Raghavan P (2009) On compressing social networks. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining (SIGKDD), Paris, France, pp 219–228
20. Claude F, Navarro F (2010) Extended compact web graph representations. In: Algorithms and applications. Lecture notes in computer science 6060. Springer, Berlin, pp 77–91
21. Claude F, Navarro G (2010) Fast and compact web graph representations. ACM Trans Web 4(4):16
22. Claude F, Navarro G (2008) Practical rank/select queries over arbitrary sequences. In: Proceedings of the 15th international symposium on string processing and information retrieval (SPIRE), Melbourne, Australia, pp 176–187
23. Claude F, Ladra S (2011) Practical representations for web and social graphs. In: Proceedings of the 20th ACM conference on information and knowledge management (CIKM), Glasgow, UK, pp 1185–1190
24. Clark D (1996) Compact Pat trees. Ph.D. Thesis, University of Waterloo, Canada
25. Demetrescu C, Finocchi I, Ribichini A (2006) Trading off space for passes in graph streaming problems. In: Proceedings of the 17th ACM-SIAM symposium on discrete algorithms (SODA), Miami, FL, pp 714–723
26. Donato D, Millozzi S, Leonardi S, Tsaparas P (2005) Mining the inner structure of the web graph. In: Proceedings of the 8th workshop on the web and databases (WebDB), Baltimore, MD, pp 145–150
27. Dourisboure Y, Geraci F, Pellegrini M (2007) Extraction and classification of dense communities in the web. In: Proceedings of the 16th international conference on world wide web (WWW) Banff, Alberta, Canada, pp 461–470
28. Gibson D, Kumar R, Tomkins A (2005) Discovering large dense subgraphs in massive graphs. In: Proceedings of the 31st international conference on very large data bases (VLDB), Trondheim, Norway, pp 721–732

29. González R, Grabowski S, Mäkinen V, Navarro G (2005) Practical implementation of rank and select queries. In: Poster Proceedings of the volume of 4th workshop on efficient and experimental algorithms (WEA), Santorini Island, Greece, pp 27–38
30. Golynski A, Munro J, Rao S (2006) Rank/select operations on large alphabets: a tool for text indexing. In: Proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithms (SODA), Miami, FL, pp 368–373
31. Grabowski S, Bieniecki W (2010) Tight and simple web graph compression. CoRR abs/006.0809
32. Grabowski S, Bieniecki W (2011) Merging adjacency lists for efficient web graph compression. Adv Intell Soft Comput 103(1):385–392
33. Grossi R, Gupta A, Vitter J (2003) High-order entropy-compressed text indexes. In: Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, MD, pp 841–850
34. Hasan M, Salem S, Zaki M (2011) SimClus: an effective algorithm for clustering with a lower bound on similarity. Knowl Inf Syst 28(3):665–685
35. Hernández C, Navarro G (2011) Compression of web and social graphs supporting neighbor and community queries. In: Proceedings of the 6th ACM workshop on social network mining and analysis (SNAKDD), San Diego, CA
36. Hernández C, Navarro G (2012) Compressed representation of web and social networks via dense subgraphs. In: Proceedings of the 19th international symposium on string processing and information retrieval (SPIRE), Cartagena de Indias, Colombia, pp 264–276
37. Katarzyna M, Przemyslaw K, Piotr B (2009) User position measures in social networks. In: Proceedings of the 4th ACM workshop on social network mining and analysis (SNAKDD), Paris, France, pp 1–9
38. Kleinberg J (1999) Authoritative sources in a hyperlinked environment. JACM 46(5):604–632
39. Kumar R, Raghavan P, Rajagopalan S, Tomkins A (1999) Trawling the web for emerging cyber-communities. Comput Netw 31(11):1481–1493
40. Larsson N, Moffat A (1999) Offline dictionary-based compression. In: Proceedings of the data compression conference (DCC), Snowbird, Utah, pp 296–305
41. Lee V, Ruan N, Jin R, Aggarwal C (2010) A survey of algorithms for dense subgraph discovery. Manag Min Graph Data 2010:303–336
42. Macropol K, Singh A (2010) Scalable discovery of best clusters on large graphs. PVLDB J 3(1):693–702
43. Maserrat H, Pei J (2010) Neighbor query friendly compression of social networks. In: Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining (SIGKDD), Washington, DC, pp 533–542
44. Mcpherson J, Ma K, Ogawa M (2005) Discovering parametric clusters in social small-world graphs. In: Proceedings of the ACM symposium on applied computing, Santa Fe, New Mexico, USA
45. Mislove A, Marcon M, Gummadi P, Druschel P, Bhattacharjee B (2007) Measurement and analysis of online social networks. In: Proceedings of the internet measurement conference (IMC), San Diego, CA, pp 29–42
46. Mishra R, Shukla S, Arora D, Kumar M (2011) An effective comparison of graph clustering algorithms via random graphs. Int J Comput Appl 22(1):22–27
47. Morik K, Kaspari A, Wurst M (2012) Multi-objective frequent termset clustering. Knowl Inf Syst 30(3):715–738
48. Randall K, Stata R, Wiener J, Wickremesinghe R (2002) The link database: fast access to graphs of the web. In: Proceedings of the data compression conference (DCC), Snowbird, UT, pp 122–131
49. Raman R, Raman V, Rao S (2002) Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the 13th annual ACM-SIAM symposium on discrete algorithms (SODA), San Francisco, CA, pp 233–242
50. Saito H, Toyoda M, Kitsuregawa M, Aihara K (2007) A large-scale study of link spam detection by graph algorithms. In: Proceedings of adversarial information retrieval on the web (AIRWeb), Banff, Alberta, Canada
51. Saito K, Kimura M, Ohara K, Motoda H (2012) Efficient discovery of influential nodes for SIS models in social networks. Knowl Inf Syst 30(3):613–635
52. Suel T, Yuan J (2001) Compressing the graph structure of the web. In: Proceedings of the data compression conference (DCC), Snowbird, UT, pp 213–222
53. Suri S, Vassilvitskii S (2011) Counting triangles and the curse of the last reducer. In: Proceedings of the 20th international conference on the world wide web (WWW), Hyderabad, India, pp 607–614
54. Van Dongen, S (2000) Graph clustering by flow simulation. Ph.D. Thesis, University of Utrecht, The Netherlands
55. Van Dongen S (2008) Graph clustering via a discrete uncoupling process. SIAM J Matrix Anal Appl 30(1):121–141

56. Vitter J (2001) External memory algorithms and data structures: dealing with massive data. ACM Comput Surv 33(2):209–271
57. Zhuge H (2009) Communities and emerging semantics in semantic link network: discovery and learning. IEEE Trans Knowl Data Eng 21(6):785–799

Author Biographies

Cecilia Hernández is currently a PhD. student at the University of Chile, Santiago, Chile. Her research interest include compressed structures, data mining, and parallel and distributed algorithms.

Gonzalo Navarro is currently full professor at the University of Chile. His areas of interest include algorithms and data structures, text searching, compression, and metric space searching. He is member of the Steering Committee of LATIN and SISAP conferences, and of the Editorial Board of Information Systems, Information Retrieval, and ACM Journal of Experimental Algorithmics.