Compact structure for sparse undirected graphs based on a clique graph partition

Felipe Glaria[a], Cecilia Hernández[b,a,*], Susana Ladra[c], Gonzalo Navarro[b,d,e], Lilian Salinas[a]

[a] *Department of Computer Science, University of Concepcion, Concepción, Chile*
[b] *CeBiB — Center for Biotechnology and Bioengineering, Chile.*
[c] *Universidade da Coruña, Centro de investigación CITIC, A Coruña, Spain*
[d] *IMFD — Millennium Institute for Foundational Research on Data, Chile*
[e] *Department of Computer Science, University of Chile, Santiago, Chile*

Abstract

Compressing real-world graphs has many benefits such as improving or enabling the visualization in small memory devices, graph query processing, community search, and mining algorithms. This work proposes a novel compact representation for real sparse and clustered undirected graphs. The approach lists all the maximal cliques by using a fast algorithm and defines a clique graph based on its maximal cliques. Further, the method defines a fast and effective heuristic for finding a clique graph partition that avoids the construction of the clique graph. Finally, this partition is used to define a compact representation of the input graph. The experimental evaluation shows that this approach is competitive with the state-of-the-art methods in terms of compression efficiency and access times for neighbor queries, and that it recovers all the maximal cliques faster than using the original graph. Moreover, the approach makes it possible to query maximal cliques, which is useful for community detection.

Keywords: graph compression, clustering, compact data structures, network analysis, maximal cliques

*Corresponding author

Email addresses: `fglaria@udec.cl` (Felipe Glaria), `cecihernandez@udec.cl` (Cecilia Hernández), `susana.ladra@udc.es` (Susana Ladra), `gnavarro@dcc.uchile.cl` (Gonzalo Navarro), `lilisalinas@udec.cl` (Lilian Salinas)

1. Introduction

A wide variety of real systems are modeled by graphs, including communication, transit, web, social, and biological networks [1, 2]. The process of discovering relevant information from graphs is called graph mining [3]. This is usually a time-consuming task, especially with the current trend of data growth size [4]. The main challenges are triggered by different aspects. This includes the data volume itself, data complexity (i.e., many relationships among the data), and application needs [4]. Several schemes have been proposed for analyzing graphs that aim at understanding the properties and patterns found in them to serve different application purposes. Some known applications include disease analysis [5], community discovery [1, 2, 6], recommender systems [7], graph compression [8, 9, 10], measuring relevance of network actors [11, 12], and network visualization [13, 14]. Recent works on graph mining postulate that dense patterns are prominent and describe different dense substructures. Some examples include maximal cliques [15, 16], communities [17], and others [3, 9, 18, 19]. These substructures have been used for improving network analysis, graph compression [9, 20], and visualization [13].

Given the space required to store and analyze large graphs, the research community has proposed graph compression formats that support basic navigation queries directly over the compressed structure without requiring decompression. This approach enables the simulation of any graph algorithm in the main memory, requiring less space than plain representations. Even though these compressed structures are usually slower than uncompressed representations, they are still attractive in devices with limited memory. This includes devices, such as tablets or cell phones. Moreover, these in-memory representations can provide faster access than plain representations incurring I/O costs [21, 22].

Although there are different types of real-world graphs of interest, this work aims at processing highly clustered and sparse graphs. Clustered graphs contain vertices grouped in highly connected subgraphs. These graphs have high clustering coefficient and transitivity [23] measures. In practice, many real-world graphs are sparse, such as graphs with low degeneracy [16].

This work proposes a compact data structure for clustered sparse undirected graphs that exploits the cliques to represent the edges implicitly. Further, it makes use of the vertex redundancy of the cliques by partitioning them into components that share many vertices. This structure enables neighbor queries, as well as queries for recovering all or subsets of the maximal cliques. Finding maximal cliques is an important step in the clique percolation method (CPM). This has been successfully used for community searches in biological networks [1], social group evolution [24], human disease pattern discovery [5], and computing and visualizing topological features using persistence homology in network analysis [14].

The structure is built on a partition of the clique graph, where each node is a maximal clique in the original graph. The proposed method uses a fast algorithm for listing all maximal cliques and defines an effective heuristic for finding a clique graph partition avoiding the construction of the clique graph. From this, a compact representation of the partitioned clique graph is proposed.

The experimental evaluation shows that the compressed graph representation is competitive with the state-of-the-art methods in terms of compression efficiency for large real graphs, obtaining the smallest representation for clustered graphs. This high compression is achieved, in some cases, at the expense of slower access times when answering neighbor queries. As discussed, in a context of limited memory or steep memory hierarchies, using less space can be of special interest. This may allow the representation to fit into faster memory levels and, in the case of larger datasets, prevents it from being handled on slower ones, such as disks [21, 22]. In addition, according to our knowledge, beside neighbor queries, the structure presented in this study is the first proposal that enables maximal clique queries. This is an important operation for applications that use clique communities. Furthermore, retrieving maximal cliques from the compressed representation is much faster than listing them from the original graph.

The implementation of the proposed method is available at http://www.inf.udec.cl/~chernand/sources/cliquecomp/cliquecomp.tgz.

2. Related work

Boldi and Vigna [25] proposed in 2004 one of the best-known techniques for web graph compression, which offered the best space/time trade-off for many years. They presented the WebGraph framework, which obtains very compact representations of web graphs by exploiting their regularities and statistical properties. More concretely, they exploit the locality of reference, since web pages generally include links to other web pages of the same domain. They also exploited the similarity of the adjacency lists or the "copy property", since the web pages from the same domain usually tend to have the same links. In addition to achieving very compact representations, this method allows for fast access to any adjacency list and offers different trade-offs depending on the desired navigation performance. The WebGraph framework outperformed many techniques that were previously proposed, which exploited the statistical properties of the graphs [26, 27, 28].

In the last decade, different authors addressed this problem using alternative approaches. Claude and Navarro [29] used a phrase-based compressor, Re-Pair, to capture the regularities present in the adjacency lists in order to compress the graph. Asano et al. [30] obtained very compact space by capturing patterns of the adjacency matrix, such as horizontal, vertical, and diagonal runs. Buehrer and Chellapilla [8] proposed the Virtual Node Miner method, which finds bicliques inside the web graph and represents them in a compact way. More concretely, for each biclique, a new virtual node is generated and all of the links within the biclique are replaced to edges to/from this new node. This significantly reduces the number of represented edges. Grabowski and Bieniecki [31] proposed a method based on list merging. This method compactly represents sets of consecutive adjacency lists with bitmaps while exploiting the similarities among those lists.

Apostolico and Drovandi [32] proposed a technique that combines the reordering of the nodes of the graph following a Breadth-First Search (BFS) strategy, which improves the locality, and a new family of universal codes for integers that follows power law distribution with an exponent close to one. This technique requires little space (about one to four bits per edge) while maintaining a retrieval time comparable to that of the WebGraph framework.

Brisaboa et al. proposed the k^2-tree method [20], which focuses on the compression of the adjacency matrix. They proposed a succinct representation for a special kind of tree inspired in region quad-trees. More concretely, the k^2-tree subdivides the adjacency matrix into $k \times k$ submatrices, which are further subdivided in case they are non-zero. This subdivision is represented in a tree following a Z-order and is then compactly represented using bitmaps. Unlike other methods, k^2-trees support the efficient retrieval of predecessors and range queries in addition to the retrieval of the original adjacency lists.

Most of the previous methods that were designed for web graphs were also successfully applied to other types of graphs such as social networks. For instance, Boldi et al. presented an improvement of WebGraph framework, based on vertex ordering, which also works well in this domain [33]. Hernández and Navarro [9] addressed the compression of web and social graphs by combining several techniques that include virtual nodes, k^2-trees, and node ordering. In addition, some proposals were specifically designed for social networks. Chierichetti et al. [34] proposed a compression scheme that takes advantage not only of the locality of reference and similarity, but also of the reciprocity, a property that is common in social networks. Maserrat and Pei [35] proposed the linearization of the input graph while exploiting regularities such as clustering.

Recently, Rossi and Zhou presented GraphZIP [36], which decomposes a graph into a set of large cliques, and then compresses and represents the graph succinctly. This clique-based method can be used as a disk-resident or as an in-memory graph representation and it is scalable for large datasets; it includes a parallel implementation.

Maximal cliques have also been used for identifying communities. For instance, clique communities have been effectively used for discovering patterns in human diseases [5] and finding influential communities in social networks [2]. Pournoor et al. [5] considered maximal cliques in protein-protein networks and found common super cliques inside disease families. A well-known algorithm for identifying clique communities is the Clique Percolation Method (CPM) [1]. This method first lists all of the maximal cliques and later builds a clique-clique overlap matrix. This is used to retrieve clique communities formed by overlapping cliques of size k. This algorithm has been made available as a tool for the most common programming languages, for instance, it has been included into R packages and Python libraries.

3. Proposed method

This section describes a new method for compressing real sparse undirected graphs using a compact data structure that takes advantage of the vertex redundancy of the graph represented by its maximal cliques. In this method, vertex redundancy refers to vertices that belong to multiple maximal cliques. Such vertices can be stored only once to reduce space.

The proposed compression method includes three steps. The first step (*clique listing*) lists all the maximal cliques of size at least two in the input graph. The second step (*clique graph partitioning*) creates a clique graph based on the maximal cliques of the graph and a partition-based clustering heuristic for the clique graph representation, without building the clique graph. The last step (*compact graph representation*) defines a compact data structure based on the symbol and bit sequences for the clique graph partitions found during the second step.

3.1. Clique listing

Let $G(V, E)$ be a graph where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. For any vertex, $u \in V$, let $N(u) = \{v \in V | (u, v) \in E\}$ be its neighborhood. A *clique* is a complete subgraph of $G(V, E)$, and we regard it as a set of vertices. In other words, every pair of distinct elements in a clique is connected by an edge. A *maximal clique* is a clique that cannot be extended by including an additional vertex. The set of all maximal cliques with at least two nodes contains all the edges in E and thus is called an *edge clique cover* of $G(V, E)$. We obtained them using the fast algorithm proposed by Eppstein and Strash [16].

3.2. Clique graph partitioning

Let $\mathcal{C} = \{c_1, c_2, \ldots, c_\mu\}$ be the collection of all of the maximal cliques of an undirected graph $G(V, E)$. The graph of cliques, called the *clique graph*, is defined such that each vertex in the graph is a maximal clique and there is an edge between two vertices if the corresponding cliques share nodes of the original graph [37, 38]. The formal definition for a clique graph is as follows.

Definition 1. Clique graph
Given a graph $G = (V, E)$ and $\mathcal{C} = \{c_1, c_2, \ldots, c_\mu\}$ being the collection of maximal cliques that cover G, the clique graph $CG_c = (V_c, E_c)$ of G is defined as

1. $V_c = \mathcal{C}$
2. $\forall c, c' \in \mathcal{C}, (c, c') \in E_c \iff c \cap c' \neq \emptyset$

Definition 2. Clique graph partition
Given a clique graph $CG_c = (V_c, E_c)$, a clique graph partition $\mathcal{CP} = \{cp_1, cp_2, \ldots, cp_M\}$ is a partition of the vertices of V_c into pairwise disjoint subsets cp_i.

The problem of finding a good clique graph partition is defined as follows:

Problem 1. Find a graph partition of the vertices in the clique graph CG_c.
Given a clique graph $CG_c = (V_c, E_c)$, output a clique graph partition $\mathcal{CP} = \{cp_1, cp_2, \ldots, cp_M\}$, aiming to maximize the shared nodes of V within the partitions while minimizing the sum of the number of partitions in which each node appears.

By definition, different partitions cannot share maximal cliques. However, it is possible to have a subset of vertices of the original graph G belonging to different partitions in the clique graph partition.

The proposed method aims at finding a clique graph partition that takes advantage of the vertex redundancy in maximal cliques. The basic goal of applying a partitioning scheme is to group maximal cliques that share many vertices, so that different partitions of the maximal cliques share none or only a few vertices.

The problem of finding a clique graph partition has been addressed recently [39]. A naive approach to find a clique graph partition is to first evaluate the similarity between all of the maximal cliques in CG_c using a method such as SimRank [40]. Afterwards, a clustering algorithm, such as spectral clustering [41] or hierarchical clustering [42], is applied. However, evaluating the similarity between the maximal cliques is time-consuming for large graphs [39]. In addition, using spectral clustering often yields separate clusters consisting of low-degree vertices [39]. The new method by Lu et al. [39] introduces a balancing factor to avoid this problem; however, as discussed by the authors, this solution is very time-consuming ($O(n^3)$), which makes it applicable for very small graphs only.

Instead, we propose an effective heuristic for finding a clique graph partition that is based on using a ranking function without the need of building the clique graph.

3.2.1. Algorithm for finding a clique graph partition

This section presents our partitioning clustering algorithm for a clique graph, which is effective for a compact representation of the input graph G.

This approach first defines a *ranking function* for each vertex in G. A ranking function assigns a score to each vertex based on some properties of the clique graph. The ranking functions consider the number and sizes of the maximal cliques where a vertex in G belongs.

The clustering heuristic is given in Algorithm 1. The output of the ranking

Algorithm 1 Clique graph partition algorithm.

```
Input: Graph $G = (V, E)$, maximal clique collection $\mathcal{C}$, ranking function $r(u)$
Output: Returns clique graph partition $\mathcal{CP}$
 1: $(C, R) \leftarrow computeRanking(r, \mathcal{C})$
 2: Initialize bit array $Z$ of size $|\mathcal{C}|$ and set each bit to 0
 3: for $u \in V$ in decreasing order of score in $R[u]$ do
 4:     $cpid \leftarrow \emptyset$
 5:     for $id \in C[u]$ do
 6:         if $Z[id] = 0$ then
 7:             $Z[id] \leftarrow 1$
 8:             $cpid \leftarrow cpid \cup \{id\}$
 9:     if $cpid \neq \emptyset$ then
10:         $\mathcal{CP} \leftarrow \mathcal{CP} : cpid$
11: return $\mathcal{CP}$
```

computation are the arrays C and R (line 1). Array C contains a list with the clique ids where each vertex in G is found, and the array R contains a score for each vertex in G. The time complexity of the ranking computation includes first passing through all of the vertices of G in the set $\mathcal{C}$ of maximal cliques, and then sorting R from highest to lowest score. The total time complexity is $O(L \log L)$, where $L = \sum_{c_i \in \mathcal{C}} |c_i|$ (i.e., all vertices in all of the maximal cliques).

Next, the array R is examined in decreasing score order. For each vertex u, we collect from $C[u]$ all the cliques where u belongs that are not already in previous partitions (we use Z to mark this). Those cliques we collect, ni *cpid*, form the next partition. The time complexity of this step is $O(|\mathcal{C}| + |V|)$. The algorithm finally returns the clique graph partition in the collection $\mathcal{CP}$, where each partition is represented as a set of clique ids.

Definition 3. Ranking function

Given a graph G and its clique collection $\mathcal{C}$, a ranking function is a function $r : V \rightarrow \mathbb{R}_{>0}$ that gives a rank score for each vertex $u \in V$.

The ranking functions are defined for each vertex based on the number and sizes of the maximal cliques where the vertex is found. Denoting $C(u) = \{c \in \mathcal{C} | u \in c\}$ for a vertex $u \in V$, we consider the following possible ranking functions.

$$r_f(u) = |C(u)| \tag{1}$$

$$r_c(u) = \sum_{c \in C(u)} |c| \tag{2}$$

$$r_r(u) = \frac{r_c(u)}{r_f(u)} \tag{3}$$

3.3. Compact graph representation

This section describes a compact data structure for representing G using the clique graph partition $\mathcal{CP}$ that was obtained in the previous step. The final representation contains compact data structures for symbol and bit sequences that include support for *rank*, *select*, and *access* operations: $rank_v(B, i)$ is the number of occurrences of bit/symbol v in $B[1, i]$, $select_v(B, j)$ is the position of the j-th occurrence of the bit/symbol v in B, and $access(B, k)$ is the bit/symbol at position k in B. The three operations can be computed in time logarithmic on the alphabet size of B by adding sublinear space on top of B [21].

The graph representation is composed of two sequences, X and Y, a bitmap B, and a byte sequence BB. For each partition $cp_p \in \mathcal{CP}$, a sequence X_p is created to store the vertices of G that are present in any of the cliques in cp_p. The sequences X_p are concatenated to obtain the sequence X. Bitmap B is used to mark the starting position of each partition in X. In addition, a byte sequence BB is implemented to register, for each vertex in X, the maximal cliques where it participates within the corresponding partition. More concretely, for each partition cp_p, a matrix of bits BB_p is created where each row represents a vertex u in the partition in the same order as in X_p, and each column represents a clique in cp_p. The columns of BB_p corresponding to the cliques where u participates are marked with 1. Each bit matrix BB_p is then converted into a byte-aligned sequence and all the sequences BB_p are concatenated to create the byte sequence

BB. If there is only one maximal clique in a partition in cp_p, then BB stores no bytes for that partition. Finally, sequence Y stores the position in BB where each partition BB_p starts. The data structure is described as follows.

Definition 4. Compact representation of $G(V, E)$

Let $\mathcal{CP} = \{cp_1, \ldots, cp_M\}$, $cp_p \in \mathcal{CP}$, and $cp_p = \{c_1, ..., c_{m_p}\}$, where m_p is the number of maximal cliques in partition cp_p. Let $bpu_p = \lceil \frac{m_p}{8} \rceil$ be the number of bytes per row of BB_p, except that $bpu_p = 0$ if $m_p = 1$ (say there are M' partitions with $bpu_p > 0$). We then define X, B, BB, and Y as follows:

$$X_p = \{u \in c, c \in cp_p\} = \{u_1, ...u_{|X_p|}\} \tag{4}$$

$$B_p = 1 : 0^{|X_p|-1} \tag{5}$$

$$BB_p = BB_p[1..|X_p|][1..bpu_p] \text{ if } m_p > 1, \text{ empty otherwise} \tag{6}$$

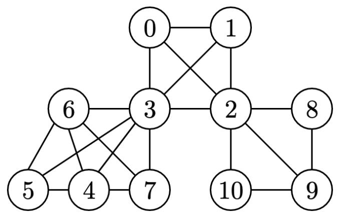
$$BB_p[i][j] = \sum_{k=1}^{8} 2^{k-1} \cdot (u_i \in c_{8(j-1)+k})$$

$$X = X_1 \cdots X_M, B = B_1 \cdots B_M, BB = BB_1 \cdots BB_{M'} \tag{7}$$

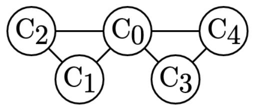
$$Y[p] = |X_{p-1}| \cdot bpu_{p-1} + Y[p-1], \qquad Y[0] = 0 \tag{8}$$

Figure 1 shows an example of the three steps in creating the data structure. The first step finds five maximal cliques in the 11-vertex input graph. Maximal cliques and the clique graph are also displayed in the top-center of the figure. The second step uses Algorithm 1 to compute the clique graph partition. The figure shows the content of the ranking array R using the function r_r and the resulting clique graph partition stored in $\mathcal{CP}$. As seen in the example, partitions with more than one maximal clique are located first in X to reduce the space of Y. The first partition in $\mathcal{CP}$ contains two maximal cliques, C_1 and C_2, the second partition contains C_3 and C_4, and the final partition contains only the maximal clique C_0. The third step builds the compact representation; Figure 1 shows the content for X, B, BB, and Y. Array X contains the vertices in all of the partitions in $\mathcal{CP}$. The first vertex in each partition in X is marked in bitmap B with a bit set to 1. For this example, BB contains bytes for the first two

Step1



$C_0 : 0, 1, 2, 3$
$C_1 : 3, 4, 6, 7$
$C_2 : 3, 4, 5, 6$
$C_3 : 2, 8, 9$
$C_4 : 2, 9, 10$



Step2

$u \in G$	0	1	3	4	5	6	7	2	8	9	10
R_r	4.0	4.0	4.0	4.0	4.0	4.0	4.0	3.3	3.0	3.0	3.0

$\mathcal{CP}$:	C_1 C_2	C_3 C_4	C_0

Step3

X:	**3**	4	5	6	7	**2**	8	9	10	**0**	1	2	3
B:	**1**	0	0	0	0	**1**	0	0	0	**1**	0	0	0
BB:	***3***	*3*	*2*	*3*	*1*	***3***	*1*	*3*	*2*				
Y:	0	5	9										

Figure 1: Example of our compression method. The first step enumerates all the maximal cliques. The second step computes the clique graph partition. Finally, the last step (bottom part of the figure) obtains the compact representation using sequences X, B, BB, and Y.

partitions. Since the partitions have two cliques, one byte per vertex is enough for representing the clique ids where each node participates. As a result, BB contains five bytes in the first partition and four bytes in the second. Given that in the first partition, vertices 3, 4, and 6 participate in both cliques C_1 and C_2, their corresponding byte at BB encodes "11" as a byte with a value of *3*. Meanwhile, vertex 7 is included only in clique C_1; therefore, its byte encodes "01" as a byte with a value of *1*. Further, vertex 5 is included only in clique C_2; therefore, its corresponding byte at BB encodes "10" as the byte with a value of *2*. A similar encoding is performed in the second partition. Finally, sequence Y indicates the starting position of each partition in BB.

4. Query algorithms

This section describes how the main queries are solved using the compact data structure. Algorithm 2 displays a sequential algorithm that retrieves the input graph G in a single pass. The time complexity of the sequential algorithm is $O(\sum_{p=1}^{M} |X_p|^2 \cdot (1 + bpu_p))$.

The algorithm goes through each partition p of the compact representation, retrieves all of the edges in that partition and adds all those edges to build E. If a partition X_p contains only one clique, then all the possible edges are generated. Otherwise, two vertices are neighbors if both participate in some clique of cp_p. To compute this, the algorithm checks if the *bitwise and* between the bitarrays of BB_p of those vertices is nonzero.

Algorithm 3 retrieves the vertices adjacent to a given $u \in V$ (i.e., its neighbors). The time complexity for retrieving the neighbors of a vertex is the same as above, but the sum ranges only over the partitions where u participates. We use *rank* and *select* on sequence X and bitvector B to find those partitions. Note that the same vertex can be inserted multiple times in $N(u)$, so we must use a set data structure that avoids repetitions.

Algorithm 4 retrieves the maximal cliques of G in time $O(\sum_{p=1}^{M} |X_p| \cdot (1 + bpu_p))$. It iterates over each partition and, inside each, it distributes every node into all the clusters where it belongs.

Let us illustrate the neighbor query algorithm using the example of Figure 1. First, let us find the neighbors of node 3. Algorithm 3 first counts the number of partitions where node 3 occurs, $occur = rank_3(X, 12) = 2$. The two partitions are then examined to find the neighbors within each. For the first partition, the position where node 3 appears is computed along with the identifier of the partition, its starting and ending positions, the number of bytes used for each element at that position, and its elements ($u_p = select_3(X, 1) = 0$, $p = rank_1(B, 0) = 1$, $s = select_1(B, 1) = 0$, $e = select_1(B, 2) - 1 = 4$, $X_p = [3, 4, 5, 6, 7]$, $bpu_p = \frac{Y[1]-Y[0]}{4-0+1} = 5/5 = 1$). Then, for each element of X_p, the algorithm checks if it is a neighbor of node 3 or not. This is done by testing if they share at least one clique of that partition, by performing the bitwise *and* with a byte value of *3*, which is the BB element of node 3 in this partition. It omits $X_p[0] = 3$, then it checks $X_p[1] = 4$, and since its associated BB element is 3, and the bitwise *and* with 3 is non-zero, then 4 is a neighbor of

Algorithm 2 Retrieve the complete list of edges E.

```
Input: $X$, $B$, $BB$, $Y$
Output: The edge set $E$
  $E \leftarrow \emptyset$
  for $p = 1$ to $|Y| - 1$ do
    $s \leftarrow select_1(B, p)$
    $e \leftarrow select_1(B, p+1) - 1$
    $bpu_p \leftarrow \frac{Y[p]-Y[p-1]}{e-s+1}$
    $X_p \leftarrow X[s..e]$
    for $j = 0$ to $|X_p| - 1$ do
      for $k = j + 1$ to $|X_p| - 1$ do
        for $b = 1$ to $bpu_p$ do
          if $BB_p[bpu_p \cdot j + b]$ & $BB_p[bpu_p \cdot k + b] \neq 0$ then
            $E \leftarrow E \cup \{(X_p[j], X_p[k])\}$ (unoriented edge)
            break
  for $p = |Y|$ to $rank_1(B, |B|)$ do
    $s \leftarrow select_1(B, p)$
    $e \leftarrow select_1(B, p+1) - 1$
    $X_p \leftarrow X[s..e]$
    for $j = 0$ to $|X_p| - 1$ do
      for $k = j + 1$ to $|X_p| - 1$ do
        $E \leftarrow E \cup \{(X_p[j], X_p[k])\}$ (unoriented edge)
  return $E$
```

3. The same happens for the rest of the elements of X_p. Therefore, 3 is a neighbor of 4, 5, 6, and 7 in this partition. Then, the second partition where 3 appears is examined. That is, $u_p = select_3(X, 12) = 0$, $p = rank_1(B, 12) = 3$, $s = select_1(B, 3) = 9$, $e = select_1(B, 4) - 1 = 12$, $X_p = [0, 1, 2, 3]$. Since $p = 3$ is equal to $|Y| = 3$, all of the nodes in this partition are in the same clique; therefore, 3 is a neighbor of all of these nodes: 0, 1, and 2. The answer of the query is then $N(3) = \{0, 1, 2, 4, 5, 6, 7\}$.

Algorithm 3 Retrieve the neighbors $N(u)$ of vertex $u \in V$.

```
Input: $u$, $X$, $B$, $BB$, $Y$
Output: The set $N(u)$
  $N(u) \leftarrow \emptyset$
  $occur \leftarrow rank_u(X, |X| - 1)$
  for $i = 1$ to $occur$ do
    $u_p \leftarrow select_u(X, i)$
    $p \leftarrow rank_1(B, u_p)$
    $s \leftarrow select_1(B, p)$
    $e \leftarrow select_1(B, p + 1) - 1$
    $X_p \leftarrow X[s..e]$
    if $p \geq |Y|$ then
      for $j = 0$ to $|X_p| - 1$ do
        if $X_p[j] \neq u$ then
          $N(u) \leftarrow N(u) \cup \{X_p[j]\}$
    else
      $bpu_p \leftarrow \frac{Y[p]-Y[p-1]}{e-s+1}$
      for $j = 0$ to $|X_p| - 1$ do
        if $X_p[j] \neq u$ then
          for $b = 1$ to $bpu_p$ do
            if $BB_p[bpu_p \cdot j + b]$ & $BB_p[bpu_p \cdot (u_p - s) + b] \neq 0$ then
              $N(u) \leftarrow N(u) \cup \{X_p[j]\}$
              break
  return $N(u)$
```

Let us now find the neighbors of node 8. This node appears in one partition only, since $occur = rank_8(X, 12) = 1$. This partition is then checked: $u_p = select_8(X, 1) = 6$, $p = rank_1(B, 6) = 2$, $s = select_1(B, 2) = 5$, $e = select_1(B, 3) - 1 = 8$, $X_p = [2, 8, 9, 10]$, $bpu_p = \frac{Y[2]-Y[1]}{4-0+1} = (9 - 5)/4 = 1$. The BB values that correspond to X_p are examined and, for each of them, the bitwise *and* with the BB value for 8, that is, a byte value of *1*, are computed.

Algorithm 4 Retrieve the maximal cliques of $G(V, E)$.

```
Input: $X$, $B$, $BB$, $Y$
Output: The collection $CC$ of maximal cliques
  $CC \leftarrow \emptyset$
  for $p = 1$ to $|Y| - 1$ do
    $s \leftarrow select_1(B, p)$
    $e \leftarrow select_1(B, p + 1) - 1$
    $bpu_p \leftarrow \frac{Y[p]-Y[p-1]}{e-s+1}$
    $X_p \leftarrow X[s..e]$
    for $j = 0$ to $|X_p| - 1$ do
      $cluster \leftarrow 0$
      Initialize empty sets $C[1], \ldots, C[8 \cdot bpu_p]$
      for $b = 1$ to $bpu_p$ do
        for $k = 1$ to $8$ do
          $cluster \leftarrow cluster + 1$
          if $BB_p[bpu_p \cdot j + b][k]$=1 then
            Insert vertex $X_p[j]$ to $C[cluster]$
    $CC \leftarrow CC \cup \{C[1], C[2], \ldots, C[cluster]\}$
  for $p = |Y|$ to $rank_1(B, |B|)$ do
    $s \leftarrow select_1(B, p)$
    $e \leftarrow select_1(B, p + 1) - 1$
    $CC \leftarrow CC \cup \{X_p[s..e]\}$
  return $CC$
```

In this case, nodes 2 and 9 are neighbors since their BB value is *3*. Instead, node 10 is not a neighbor, as its BB value is *2*, and the bitwise *and* with value *1* is zero. Therefore, $N(8) = \{2, 9\}$.

5. Experimental evaluation

This section describes several experiments to tune and compare our method with the state-of-the-art algorithms for compressing graphs, including version

3.6.1 of WebGraph (WG) [33], the graph compression by BFS from Apostolico and Drovandi (AD) [32], and the k^2-tree [20]. The results of the compression efficiency reported by Rossi and Zhou for GraphZIP [36] are also included, although they do not support query operations. All of the experiments ran on a machine with an Intel i7-7500U CPU @ 2.70GHz, 12 GB RAM, and 4 MB cache. We used a g++ version 8.2.1 compiler with optimization flag -O3.

Our implementation uses succinct data structures to represent the sequences of symbols and bits. Concretely, we use compact data structures based on the wavelet matrix [43] for X and Y and compressed bitmaps [44] for B. Such representations are available in the Succinct Data Structure Library (SDSL) version 2.0[1] [45]. For the byte sequence BB, plain Huffman compression [46] was used.

Real-world graphs from different sources were selected: the sparse social network graphs `dblp2010` and `dblp2011` from the Laboratory for Web Algorithmics (LAW)[2], the `snapdblp` dataset from SNAP[3], `markastro`, `markcmat2003`, `markcmat2005` from Mark Newman[4], `coPapersDBLP` and `coPapersCiteseer` from the 10th DIMACS Implementation Challenge[5], and the other graphs available from the Network Repository Project[6]. Table 1 lists the main statistics of those graphs, including the number of vertices, the number of edges, the average degree, and the maximum degree of the vertices of each graph.

It is important to note that our approach is being compared with techniques that encode directed graphs. WG and AD compress adjacency lists; therefore, their queries are focused on retrieving out-neighbors of any vertex. To make them represent undirected graphs, we duplicated the edges of the graphs. On the other hand, the k^2-tree can efficiently retrieve both in- and out-neighbors of the graphs, thus it can represent undirected graphs without duplicating edges.

[1]`https://github.com/simongog/sdsl-lite`
[2]`http://law.di.unimi.it/datasets.php`
[3]`https://snap.stanford.edu/data/`
[4]`http://www-personal.umich.ed/~mejn/netdata/`
[5]`https://www.cc.gatech.edu/dimacs10/archive/coauthor.shtml`.
[6]`https://networkrepository.com/`

Table 1: Main properties of the datasets used in the experiments.

Dataset	$\|V\|$	$\|E\|$	avg degree	max degree
ca-CondMat	21,363	182,572	8.54	279
ca-MathSciNet	332,689	1,641,288	4.93	496
ca-AstroPh	17,903	393,944	22.00	504
ca-HepPh	11,204	235,238	20.99	491
bio-wormnetv3	2,445	157,472	64.40	347
markastro	16,706	242,502	14.51	360
markcmat2003	30,460	240,058	7.88	202
markcmat2005	39,577	351,386	8.69	278
snapdblp	317,080	2,099,732	6.62	2,752
dblp2010	326,186	1,615,400	4.95	238
dblp2011	986,324	6,707,236	6.80	979
coPapersDBLP	540,486	30,491,458	56.41	3,299
coPapersCiteseer	434,101	32,073,440	73.88	1,188
coAuthorsDBLP	299,067	1,955,352	6.53	336
coAuthorsCiteseer	227,320	1,628,268	7.16	1,372
bcsstk33	8,738	583,166	66.73	140
crankseg-1	52,804	10,561,406	200.01	2,702
crankseg-2	63,838	14,085,020	220.63	3,422
mixtank-new	29,957	1,965,084	65.59	153
dblp-coauthors	1,314,050	10,724,828	8.16	1,545
ns3Da	20,414	1,660,392	81.33	305
sme3Dc	42,930	3,105,790	72.34	404

5.1. *Analysis of ranking functions*

This section discusses how the proposed method behaves depending on the ranking function chosen. Tuning this parameter allows our method achieve its best performance.

Figure 2 presents the maximal clique size distribution of the four graphs, including those that achieve the lowest and highest compression performance for the proposed method, as described later in Tables 4 and 5. The size of the cliques are as follows: in `ca-MathSciNet`, they are below 25; in `dblp-coauthors`, they are below 60; in `coPapersDBLP`, they are up to 300; and finally in `crankseg_2`, they are between 50 and 150. This aspect is important in the compression efficiency, because when cliques are larger, more edges can be represented implicitly, thus saving more space (as in the case of `coPapersDBLP` and `crankseg_2`).

The ranking functions (r_r, r_c, and r_f) given in Definition 3 were analyzed to determine the best options for the compressed representation.

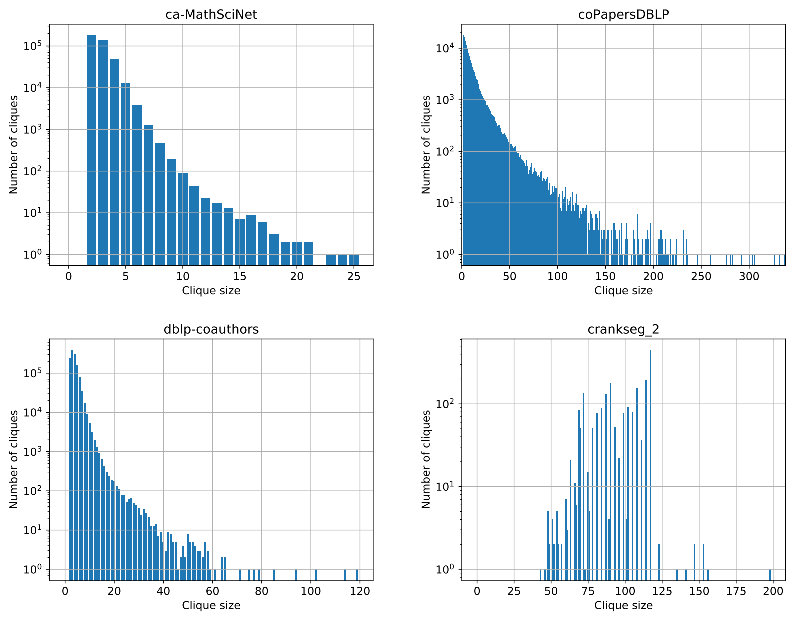


Figure 2: Clique size distribution for four datasets.

5.2. Compression performance

Figure 3 shows the space usage, in bits per edge (bpe[7]), of each component of our compressed representation, using the three ranking functions. The arrays X and BB contribute most of the space. The ranking functions provide slight differences in the total number of bits in the structure, though r_f performs best in most graphs.

In general, when BB increases, X decreases. This happens because, when there are more shared vertices in a clique partition, BB requires to encode more cliques per vertex in a partition of X. Then, the number of elements in X decreases because the common vertices of the cliques in the partition are written only once. In the same way, when BB encodes fewer cliques per vertex, it requires less space and X grows.

The relation between clique sizes and compression performance is apparent. For example, Figure 2 shows that the cliques in `coPapersDBLP` are much bigger than those in `ca-MathSciNet`. As a consequence, the bpe required by the former are much less than the latter.

Table 2 shows the number of maximal cliques, the degeneracy, the clustering coefficient, and the transitivity [23] of the graphs. The clustering coefficient is a measure that indicates how nodes tend to group in a graph. Graphs with highly connected nodes have a high clustering coefficient. Transitivity measures the fraction of vertex pairs that are both connected to a third vertex. This indicates how well connected the nodes are in the network. In addition, the last two columns show the compression ratio obtained by our method (in bpe) using two ranking functions. We can see that the best compression (fewer bits per edge) is achieved in graphs `coPapersDBLP` and `coPapersCiteseer`. They have the highest clustering coefficient and the largest transitivity. The worst compression is achieved by `ca-MathSciNet`, which has the lowest clustering coefficient and transitivity. According to the results, we suggest that this approach works well for graphs that have a number of maximal cliques that is proportional to the

[7]Total number of bits in the structure divided by the number of edges in the graph.

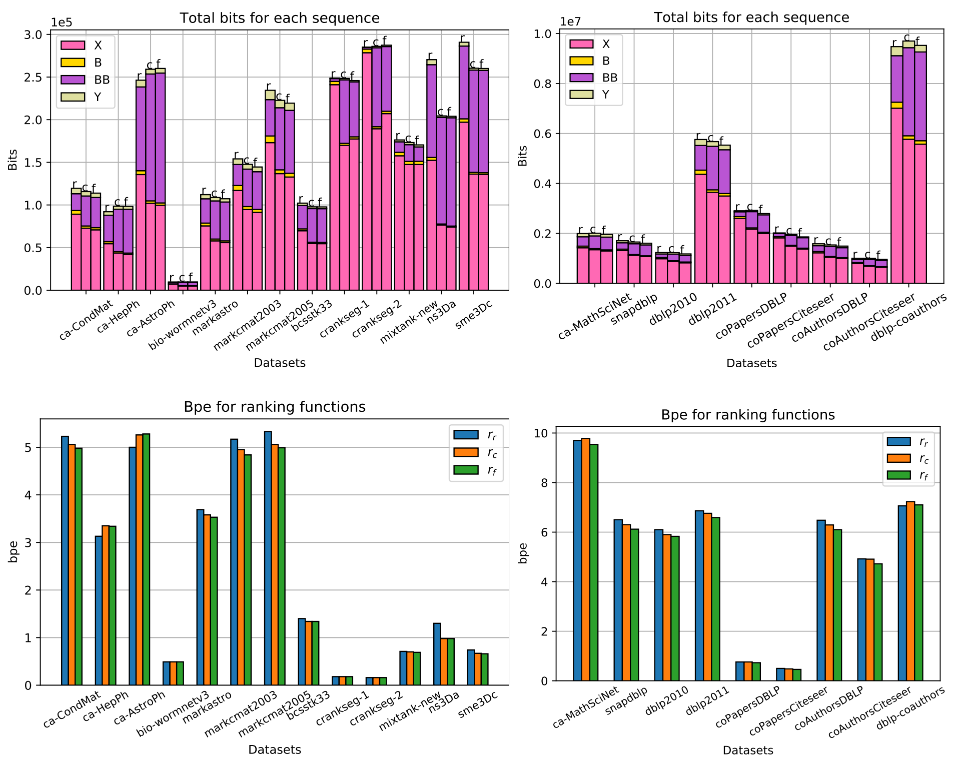


Figure 3: The total number of bits used by the components of our compact representation (top) and the space required (in bits per edge, bpe) using each ranking function over all the datasets (bottom).

number of vertices of the graph, and that are sparse and clustered, according to the degeneracy, the clustering coefficient, and the transitivity metrics.

Our structure is unique in that it enables the recovery of all the maximal cliques. Table 3 compares the time to do this from the original graph (using the algorithm of Eppstein and Strash [16]) and from our compressed representation, employing one or four threads. The results show that, even using one thread, it is faster to obtain the maximal cliques from our representation, where they are somehow preprocessed, than obtaining directly from the original graph. In addition, our algorithm is easily parallelizable, which yields even wider performance differences. We remark that other compression methods would require

Table 2: The number of maximal cliques (μ), degeneracy (degen.), clustering coefficient (CCoeff.), transitivity (Tr.), bits per edge using the ranking function r_r (bpe_r), and bits per edge using ranking function r_f (bpe_f).

Dataset	$\mu = \lvert\mathcal{C}\rvert$	degen.	CCoeff.	Tr	bpe_f	bpe_r
`ca-CondMat`	17,757	25	0.63	0.26	4.98	5.23
`ca-MathSciNet`	387,758	24	0.41	0.14	9.54	9.70
`ca-AstroPh`	36,084	56	0.63	0.32	5.28	5.00
`ca-HepPh`	14,588	238	0.61	0.65	3.34	3.13
`bio-wormnetv3`	528	125	0.83	0.72	0.49	0.49
`markastro`	15,724	56	0.66	0.42	3.53	3.69
`markcmat2003`	23,117	24	0.63	0.28	4.84	5.17
`markcmat2005`	34,274	29	0.64	0.24	4.99	5.33
`snapdblp`	257,551	113	0.63	0.30	6.12	6.50
`dblp2010`	196,434	74	0.61	0.39	5.83	6.10
`dblp2011`	806,320	118	0.63	0.20	6.59	6.86
`coPapersDBLP`	139,340	336	0.80	0.65	0.73	0.76
`coPapersCiteseer`	86,303	844	0.82	0.77	0.46	0.50
`coAuthorsDBLP`	240,681	114	0.63	0.32	6.10	6.48
`coAuthorsCiteseer`	145,904	86	0.68	0.46	4.72	4.92
`bcsstk33`	10,260	35	0.51	0.47	1.34	1.40
`crankseq-1`	2,062	128	0.60	0.48	0.18	0.18
`crankseq-2`	2,062	197	0.62	0.49	0.16	0.16
`mixtank-new`	4,771	44	0.63	0.43	0.69	0.71
`dblp-coauthors`	1,260,601	118	0.64	0.17	7.10	7.06
`ns3Da`	21,141	50	0.59	0.44	0.98	1.30
`sme3Dc`	8,522	44	0.62	0.46	0.66	0.74

full decompression of the graph and then run the maximal clique detection algorithm on the original graph, thereby requiring much higher time and working space. Further, we know of no parallel version of the algorithm to obtain the maximal cliques [16] from the original graph. Easily supporting clique queries is one of our main contributions.

5.3. Comparison with the state of the art

This section compares our method with several state-of-the-art alternatives: the WebGraph representation (WG) [33], the graph compression by BFS from Apostolico and Drovandi (AD) [32], and the k^2-tree [20]. WG is parameterized to give support for random access. In the comparison, two variants of our method are included, namely CC$_r$ using r_r, and CC$_f$ using r_f. These are the variants that achieve the best space/time trade-off.

Table 4 displays the compression efficiency and the average time to retrieve the neighbors of 10^6 random nodes.[8] We observe that CC$_f$ obtains the best compression for more than half of the datasets, and is the second best (losing to k^2-trees) in almost all the others, except in three cases where CC$_r$ is second. Those representations always outperform WG and AD in terms of space.

In terms of time performance, WG is the fastest by far, followed in most cases by AD. Those structures use much more space than ours, however. Our times are more similar to those of the k^2-tree, which is also closer to ours in space usage. Our representation uses less space and more time than k^2-trees in 12 of the 22 datasets, and more space and less time in 6 of the 22 datasets; it loses in both aspects in the other 4.

An alternative metric, used by Rossi and Zhou [36], measures the compression as the space saving, $S_c = 1 - \frac{G_c}{G}$, where G_c is the number of bytes of the compressed graph and G is the number of bytes of the original graph. Unlike

[8]We use the same random nodes for measuring the time performance of all compression techniques. We also use the same vertex ordering (breadth-first search) for all of the graphs; therefore ensuring that the adjacency lists returned by all of the techniques are the same.

Table 3: The time for listing all the maximal cliques (in seconds) from the original graph (time orig), from the proposed compact data structure using one thread (time compr) and using four threads (time paral). The last column shows the speedup of the parallel execution with respect to the time using the original graph representation.

Dataset	$\|V\|$	$\mu = \|\mathcal{C}\|$	time orig	time compr	time paral	speed up
ca-CondMat	21,363	17,757	0.06	0.04	0.02	3.00
ca-MathSciNet	332,689	387,758	0.87	0.83	0.57	1.52
ca-AstroPh	17,903	36,084	0.24	0.11	0.05	4.80
ca-HepPh	11,204	14,588	0.14	0.06	0.02	7.00
bio-wormnetv3	2,445	528	0.06	0.012	0.003	20.00
markastro	16,706	15,724	0.18	0.05	0.02	9.00
markcmat2003	30,460	23,117	0.09	0.05	0.038	2.36
markcmat2005	39,577	34,274	0.28	0.11	0.05	5.60
snapdblp	317,080	257,551	1.68	0.70	0.43	3.90
dblp2010	326,186	196,434	1.12	0.51	0.29	3.86
dblp2011	986,324	806,320	5.58	2.20	1.43	3.90
coPapersDBLP	540,486	139,340	17.96	1.10	0.66	27.21
coPapersCiteseer	434,102	86,303	26.70	0.73	0.43	62.09
coAuthorsDBLP	299,067	240,681	0.90	0.67	0.44	2.04
coAuthorsCiteseer	227,320	145,904	0.61	0.43	0.23	2.65
bcsstk33	8,738	10,260	0.46	0.11	0.03	15.33
crankseg-1	52,804	2,062	10.26	0.10	0.04	256.50
crankseg-2	63,838	2,062	16.90	0.12	0.05	338.00
mixtank-new	29,957	4,771	1.64	0.10	0.05	32.8
dblp-coauthors	1,314,050	1,260,601	6.54	4.48	2.64	2.47
ns3Da	20,414	21,141	1.55	0.11	0.06	25.83
sme3Dc	42,930	8,522	1.82	0.08	0.04	45.50

Table 4: The compression space (in bpe) and random-access time (in μs) for the proposed structure and other state-of-the-art methods. The two best compression results for each dataset are marked in bold; the best one is underlined.

Dataset	CC$_f$		CC$_r$		WG random		k^2-tree		AD	
	space (bpe)	time (μs)	space (bpe)	time (μs)	space (bpe)	time (μs)	space (bpe)	time (μs)	space (bpe	time (μs)
`ca-CondMat`	**4.98**	2.89	5.23	2.79	10.12	0.08	**5.79**	2.49	7.53	2.32
`ca-MathSciNet`	**9.54**	5.45	9.70	5.31	15.26	0.14	**6.43**	9.20	10.11	2.52
`ca-AstroPh`	5.28	5.15	**5.00**	3.54	7.33	0.06	**4.92**	1.51	6.87	1.89
`ca-HepPh`	3.34	3.34	**3.13**	2.31	4.62	0.05	**2.98**	0.88	3.81	1.31
`bio-wormnetv3`	**0.49**	0.80	0.49	0.80	1.49	0.03	**0.70**	0.10	0.74	0.04
`markastro`	**3.53**	2.80	3.69	2.31	8.10	0.05	**4.34**	1.33	5.67	1.79
`markcmat2003`	**4.84**	2.87	5.17	2.85	11.50	0.07	**5.50**	2.58	7.27	2.26
`markcmat2005`	**4.99**	3.27	5.33	2.98	11.78	0.07	**5.60**	2.80	7.86	2.32
`snapdblp`	**6.12**	4.01	6.50	3.93	11.80	0.13	**5.23**	6.93	8.14	2.30
`dblp2010`	**5.83**	3.92	6.10	4.04	8.67	0.10	**4.30**	4.84	6.71	2.15
`dblp2011`	**6.59**	5.35	6.86	4.99	10.13	0.13	**5.89**	10.69	9.67	2.36
`coPapersDBLP`	**0.73**	1.60	0.76	1.42	2.71	0.05	**0.94**	1.16	1.81	0.80
`coPapersCiteseer`	**0.46**	1.15	0.50	1.24	1.79	0.05	**0.45**	0.49	0.85	0.45
`coAuthorsDBLP`	**6.10**	3.99	6.48	3.90	11.29	0.14	**5.16**	6.56	8.00	2.28
`coAuthorsCiteseer`	**4.72**	3.52	4.92	3.16	8.64	0.12	**3.77**	4.00	5.66	2.04
`bcsstk33`	**1.34**	0.70	1.40	0.72	1.67	0.04	**1.41**	0.14	1.71	0.94
`crankseq-1`	**0.18**	0.97	0.18	0.93	1.00	0.03	**0.25**	0.08	0.61	0.36
`crankseq-2`	**0.16**	0.96	0.16	0.89	0.93	0.03	**0.23**	0.08	0.55	0.35
`mixtank-new`	**0.69**	1.24	0.71	1.31	2.25	0.05	**1.16**	0.17	2.07	0.95
`dblp-coauthors`	7.10	7.77	**7.06**	5.92	15.73	0.10	**6.32**	13.20	10.44	2.42
`ns3Da`	**0.98**	1.73	1.30	2.02	4.15	0.06	**1.41**	0.16	2.00	1.01
`sme3Dc`	**0.66**	1.41	0.74	1.39	3.80	0.06	**1.08**	0.17	1.83	0.99

the bits-per-edge metric used in the previous tables, higher values of S_c indicate a better performance. Table 5 lists the space saving S_c for the largest undirected graphs used by Rossi and Zhou [36]. We could not obtain the compression performance for the rest of the datasets because their implementation is not available. The table shows that our structures compress much better than GraphZIP in all four datasets. The space savings of their version GraphZIP$_f$ (GraphZIP with an additional compression function [36]) is much more competitive, but our CC$_f$ still outperforms it clearly in two cases, whereas in the other two there is a positive or negative difference of about 1%.

Table 6 compares the construction times required by all the methods. Our

Table 5: The compression results measured as space savings (higher is better), considering two GraphZIP variants.

Dataset	CC$_f$	CC$_r$	WGs	AD	k^2-tree	GraphZIP	GraphZIP$_f$
`ca-CondMat`	0.8605	0.8536	0.7539	0.7893	0.8378	0.3058	0.8046
`ca-MathSciNet`	0.7521	0.7478	0.6590	0.7372	0.8328	0.2014	0.7643
`ca-AstroPh`	0.8422	0.8505	0.7981	0.7945	0.8529	0.3289	0.6589
`ca-HepPh`	0.9003	0.9066	0.8785	0.8861	0.9109	0.5769	0.8949
`bio-wormnetv3`	0.9848	0.9848	0.9595	0.9773	0.9784		
`markastro`	0.8963	0.8917	0.7858	0.8337	0.8427		
`markcmat2003`	0.8656	0.8566	0.7743	0.7985	0.8475		
`markcmat2005`	0.8594	0.8499	0.7647	0.7812	0.8419		
`snapdblp`	0.8338	0.8235	0.7239	0.7788	0.8580		
`dblp2010`	0.8462	0.8391	0.8180	0.8231	0.8867		
`dblp2011`	0.8190	0.8117	0.7609	0.7347	0.8384		
`coPapersDBLP`	0.9775	0.9766	0.9236	0.9443	0.9709		
`coPapersCiteseer`	0.9857	0.9845	0.9497	0.9735	0.9859		
`coAuthorsDBLP`	0.8346	0.8243	0.7383	0.7832	0.8601		
`coAuthorsCiteseer`	0.8706	0.8649	0.8029	0.8446	0.8965		
`bcsstk33`	0.9588	0.9569	0.9543	0.9473	0.9566		
`crankseg-1`	0.9943	0.9942	0.9711	0.9808	0.9922		
`crankseg-2`	0.9950	0.9950	0.9730	0.9828	0.9927		
`mixtank-new`	0.9787	0.9780	0.9369	0.9361	0.9641		
`dblp-coauthors`	0.8023	0.8033	0.6024	0.7093	0.8240		
`ns3Da`	0.9697	0.9598	0.8779	0.9381	0.9562		
`sme3Dc`	0.9794	0.9770	0.8893	0.9434	0.9665		

method is competitive with the k^2-tree and WG, and builds consistently faster than AD. Rossi and Zhou [36] also report the construction times of GraphZIP; however, since their implementation is not available, their running times are not included.

Finally, Table 7 shows the time required to decompress the complete graph. Our technique generally competitive and in many cases the fastest, but it is sharply outperformed by the other methods on the graphs where we obtain the best compression ratio. Since we allow querying the graph directly in compressed form, however, one can actually never decompress it.

Table 6: Construction time in seconds.

Dataset	CC_f	WG	AD	k^2-tree
`ca-CondMat`	0.18	0.46	3.07	0.11
`ca-MathSciNet`	3.01	1.57	7.13	1.01
`ca-AstroPh`	0.61	0.64	3.69	0.14
`ca-HepPh`	0.27	0.38	3.19	0.34
`bio-wormnetv3`	0.09	0.23	3.00	0.11
`markastro`	0.31	0.22	3.26	0.15
`markcmat2003`	0.24	0.42	3.09	0.09
`markcmat2005`	0.48	0.61	3.36	0.13
`snapdblp`	3.44	1.18	7.79	1.22
`dblp2010`	2.35	1.07	7.74	0.81
`dblp2011`	12.10	3.26	16.75	5.57
`coPapersDBLP`	21.23	5.28	83.88	5.68
`coPapersCiteseer`	28.80	4.56	65.85	3.50
`coAuthorsDBLP`	2.55	1.18	7.81	1.11
`coAuthorsCiteseer`	1.67	1.20	7.16	0.57
`bcsstk33`	0.65	0.53	5.83	0.17
`crankseg-1`	10.63	1.32	25.98	0.78
`crankseg-2`	17.34	1.60	30.92	1.10
`mixtank-new`	1.88	0.78	8.41	0.32
`dblp-coauthors`	16.44	6.05	26.00	11.01
`ns3Da`	2.14	0.97	8.54	0.19
`sme3Dc`	2.16	1.25	15.57	0.29

Table 7: Decompression time, in seconds, for our structure and other state-of-the-art methods. WGseq is WebGraph configured for sequential access. AD was not included since it does not have a sequential access option.

Dataset	CC$_f$	CC$_r$	WGseq	k^2-tree
`ca-CondMat`	0.072	0.078	0.347	0.056
`ca-MathSciNet`	1.121	1.120	0.990	0.350
`ca-AstroPh`	0.275	0.163	0.380	0.061
`ca-HepPh`	0.100	0.101	0.344	0.011
`bio-wormnetv3`	0.016	0.016	0.060	0.070
`markastro`	0.115	0.108	0.281	0.023
`markcmat2003`	0.090	0.110	0.040	0.060
`markcmat2005`	0.158	0.149	0.527	0.041
`snapdblp`	0.980	1.102	1.201	0.356
`dblp2010`	0.694	0.779	1.201	0.164
`dblp2011`	4.173	4.197	2.411	1.313
`coPapersDBLP`	5.140	4.318	1.596	1.012
`coPapersCiteseer`	3.471	3.696	1.563	0.658
`coAuthorsDBLP`	0.921	1.011	0.971	0.300
`coAuthorsCiteseer`	0.700	0.672	0.947	0.133
`bcsstk33`	0.090	0.100	0.330	0.014
`crankseg-1`	1.068	1.019	1.090	0.097
`crankseg-2`	1.408	1.271	1.300	0.107
`mixtank-new`	0.281	0.293	0.580	0.069
`dblp-coauthors`	10.024	7.530	1.540	1.725
`ns3Da`	0.348	0.319	0.491	0.055
`sme3Dc`	0.442	0.424	0.860	0.072

6. Conclusions

This work introduces a new compact representation of real sparse and clustered undirected graphs based on clique graph partitioning. The method first lists all the maximal cliques of the input graph. Then, it defines a clique graph, whose vertices are the cliques in the original graph. Next, it finds a partition of the clique graph, which is finally encoded in a compressed form using compact data structures.

Our method includes an effective heuristic to find a partition in the clique graph, by defining ranking functions for the vertices of the original graph. Different ranking functions were defined and their behavior was analyzed. We selected the ones that obtain the best trade-offs between space and neighbor random access time. In addition, the compressed structure supports the recovery of all or a subset of maximal cliques, which might be useful for community search algorithms. Our compressed representation reduces the time to retrieve all the maximal cliques, especially on large graphs, where our method is up to two orders of magnitude faster.

We compared our compressed representation with the best-known state-of-the-art techniques. Our structure always obtains the best or second-best space, while staying competitive in the time to retrieve the neighbor list of random nodes. We obtain the best compression ratios on graphs with a high clustering coefficient and transitivity. This trade-off can be very convenient when the significantly smaller space usage of the structure allows it to reside in the main memory while alternative representations must be deployed on disk. Finally, we show that our structure is practical in terms of compression and decompression performance of the whole graph.

In regard to future work, we will study alternative partitioning algorithms, including finding minimal edge clique covers and alternative compact data structures for improving the space/time tradeoffs. In addition, we will design more complex algorithms on top of this compact graph representation to perform different network analyses.

Acknowledgments

This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie [grant agreement No 690941]; from the Ministerio de Economía y Competitividad (PGE and ERDF) [grant numbers TIN2016-77158-C4-3-R]; from Xunta de Galicia (co-founded with ERDF) [grant numbers ED431C 2017/58; ED431G 2019/01]; from the Center for Biotechnology and Bioengineering (CeBiB), Chile; and from the Millennium Institute for Foundational Research on Data (IMFD), Chile.

References

[1] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814, 2005.

[2] Jianxin Li, Xinjue Wang, Ke Deng, Xiaochun Yang, Timos Sellis, and Jeffrey Xu Yu. Most influential community search over large social networks. In *IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 871–882, 2017.

[3] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1): 75–105, 2013.

[4] Bin Shao, Haixun Wang, and Yanghua Xiao. Managing and mining large graphs: systems and implementations. In *ACM International Conference on Management of Data (SIGMOD)*, pages 589–592, 2012.

[5] Ehsan Pournoor, Naser Elmi, Yosef Masoudi-Sobhanzadeh, and Ali Masoudi-Nejad. Disease global behavior: A systematic study of the human interactome network reveals conserved topological features among categories of diseases. *Informatics in Medicine Unlocked*, 17:100249, 2019.

[6] Bin Zhou. Applying the clique percolation method to analyzing cross-market branch banking network structure: the case of illinois. *Social Network Analysis and Mining*, 6(1):11, 2016.

[7] Lidia Fotia. Recommending items in social networks using cliques-based trust. In *WOA*, pages 51–56, 2018.

[8] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to Web graph compression with communities. In *1st ACM International Conference on Web Search and Data Mining (WSDM)*, pages 95–106, 2008.

[9] Cecilia Hernández and Gonzalo Navarro. Compressed representations for web and social graphs. *Knowledge and Information Systems*, 40(2):279–313, 2014.

[10] Natalie Stanley, Roland Kwitt, Marc Niethammer, and Peter J Mucha. Compressing networks with super nodes. *Scientific Reports*, 8(1):10892, 2018.

[11] Øivind Wang, Nicolai Bodd, Chen Xing, Bård Kvalheim, and Torbjørn Helvik. Enterprise graph search based on object and actor relationships, 2017. US Patent 9,542,440.

[12] Zhipeng Huang, Yudian Zheng, Reynold Cheng, Yizhou Sun, Nikos Mamoulis, and Xiang Li. Meta structure: Computing relevance in large heterogeneous information networks. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1595–1604. ACM, 2016.

[13] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *29th Conference on Artificial Intelligence (AAAI)*, 2015.

[14] Bastian Rieck, Ulderico Fugacci, Jonas Lukasczyk, and Heike Leitte. Clique community persistence: A topological visual analysis approach for complex

networks. *IEEE transactions on visualization and computer graphics*, 24 (1):822–831, 2017.

[15] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 260–272. Springer, 2004.

[16] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithmics*, 18:3–1, 2013.

[17] Zhiyuan Liu and Yinghong Ma. A divide and agglomerate algorithm for community detection in social networks. *Information Sciences*, 482:321–333, 2019.

[18] Charalampos Tsourakakis. The k-clique densest subgraph problem. In *24th International Conference on World Wide Web (WWW)*, pages 1122–1132, 2015.

[19] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. Scalable large near-clique detection in large-scale networks via sampling. In *21th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 815–824. ACM, 2015.

[20] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.

[21] Gonzalo Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 570 pages.

[22] Mohammad Khavari Tavana, Yifan Sun, Nicolas Bohm Agostini, and David Kaeli. Exploiting adaptive data compression to improve performance and

energy-efficiency of compute workloads in multi-gpu systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 664–674, 2019.

[23] Jari Saramäki, Mikko Kivelä, Jukka-Pekka Onnela, Kimmo Kaski, and Janos Kertesz. Generalizations of the clustering coefficient to weighted complex networks. *Physical Review E*, 75(2):027105, 2007.

[24] Gergely Palla, Albert-László Barabási, and Tamás Vicsek. Quantifying social group evolution. *Nature*, 446(7136):664, 2007.

[25] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *13th International World Wide Web Conference (WWW)*, pages 595–601, 2004.

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

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

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

[29] F. Claude and G. Navarro. Fast and compact Web graph representations. *ACM Transactions on the Web (TWEB)*, 4(4):article 16, 2010.

[30] Y. Asano, Y. Miyawaki, and T. Nishizeki. Efficient compression of Web graphs. In *14th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 5092, pages 1–11, 2008. ISBN 978-3-540-69732-9.

[31] Sz. Grabowski and W. Bieniecki. Merging adjacency lists for efficient web graph compression. In *Man-Machine Interactions 2 AISC 103*, pages 385–392, 2011.

[32] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[33] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *20th International Conference on World Wide Web (WWW)*, pages 587–596. ACM, 2011.

[34] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 219–228, 2009.

[35] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 533–542, 2010.

[36] Ryan A Rossi and Rong Zhou. GraphZIP: a clique-based sparse graph compression method. *Journal of Big Data*, 5(1):10, 2018.

[37] Ronald C. Hamelink. A partial characterization of clique graphs. *Journal of Combinatorial Theory*, 5(2):192–197, 1968.

[38] Fred S. Roberts and Joel H. Spencer. A characterization of clique graphs. *Journal of Combinatorial Theory, Series B*, 10(2):102–108, 1971.

[39] Zhenqi Lu, Johan Wahlström, and Arye Nehorai. Community detection in complex networks via clique conductance. *Scientific Reports*, 8(1):5982, 2018.

[40] Weiren Yu, Xuemin Lin, Wenjie Zhang, Jian Pei, and Julie A McCann. Simrank*: Effective and scalable pairwise similarity search based on graph topology. *The VLDB Journal*, pages 1–26, 2019.

[41] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems*, pages 849–856, 2002.

[42] Jasmine Irani, Nitin Pise, and Madhura Phatak. Clustering techniques and the similarity measures used in clustering: A survey. *International Journal of Computer Applications*, 134(7):9–14, 2016.

[43] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[44] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[45] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

[46] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.