Rossi and Zhou *J Big Data (2018) 5:10*
https://doi.org/10.1186/s40537-018-0121-z

Journal of Big Data

RESEARCH Open Access

GraphZIP: a clique-based sparse graph compression method

Ryan A. Rossi[1*] and Rong Zhou[2]

*Correspondence:
rrossi@adobe.com
[1] Adobe Research, 345 Park Ave, San Jose, CA, USA
Full list of author information is available at the end of the article

Abstract

Massive graphs are ubiquitous and at the heart of many real-world problems and applications ranging from the World Wide Web to social networks. As a result, techniques for compressing graphs have become increasingly important and remains a challenging and unsolved problem. In this work, we propose a graph compression and encoding framework called GraphZIP based on the observation that real-world graphs often form many cliques of a large size. Using this as a foundation, the proposed technique decomposes a graph into a set of large cliques, which is then used to compress and represent the graph succinctly. In particular, disk-resident and in-memory graph encodings are proposed and shown to be effective with three important benefits. First, it reduces the space needed to store the graph on disk (or other permanent storage device) and in-memory. Second, GraphZIP reduces IO traffic involved in using the graph. Third, it reduces the amount of work involved in running an algorithm on the graph. The experiments demonstrate the scalability, flexibility, and effectiveness of the clique-based compression techniques using a collection of networks from various domains.

Keywords: Graph compression, Large cliques, Graph encoding, Sparse graphs, Graph algorithms

Introduction

This work proposes a fast parallel framework for graph compression based on the notion of cliques. There are two fundamentally important challenges in the era of big graph data [1], namely, developing faster and more efficient graph algorithms and reducing the amount of space required to store the graph on disk or load it into memory [2, 3]. As such, we propose a clique-based graph compression method called GraphZIP that improves both the runtime and performance of graph algorithms (and more generally computations). Moreover, it reduces the space required to store the graph in both memory and disk and carry out computations. In addition, the techniques are shown to be effective for compressing both large sparse graphs as well as dense graphs. Both types of graphs arise frequently in many application domains and thus are of fundamental importance.

The clique-based methods support both lossless [4] and lossy graph compression [5]. For lossless compression, any heuristic or exact clique finder can be used in the clique discovery phase [6–9]. Heuristic clique finders are faster but lead to worse compression.

Depending on application domain constraints, one may trade off time and accuracy (better compression). For instance, compressing the graph is a one-time cost and for many applications it might be warranted to spend extra time computing a better compression (via an exact clique finder). These techniques are also flexible for *lossy graph compression* by relaxing the notion of clique via any arbitrary clique relaxation [10–12]. For instance, consider approximate cliques (near-cliques) that are missing a few edges (that would otherwise form a valid clique). Thus, it is easy to simply encode the missing edges as errors, and use previous techniques with the exception that the computations would need to incorporate the errors, which adds a small computational cost, but may reduce the space needed quite significantly. Clearly, the pseudo cliques must not have too many missing edges (errors), otherwise the computational costs outweigh the benefits. Most of these algorithms are easily constrained such that a pseudo clique does not contain more than k missing edges.

The applications and advantages of such graph compression techniques are endless. For instance, parallel computing architectures such as the GPU [13–15] have limited memory and thus unable to handle massive graphs [16, 17]. In addition, GPU algorithms must be designed with this limitation in mind, and in many cases, may be required to perform significantly more computations, in order to avoid using additional memory [18, 19]. For instance, checking if a neighbor exists on the CPU takes $o(1)$ using a perfect hash table whereas on the GPU it takes $\mathcal{O}(\log |N(v)|)$ using a binary search. This is because storing a perfect hash table for each worker (core, processing unit) on the GPU is impractical for most graphs.

Existing graph algorithms may leverage the proposed graph encoding with little effort (as opposed to the existing specialized techniques). For instance, it can easily replace the current graph encoding such as CSR/CSC [20] and the ilk by implementing a few graph primitive functions such as "retrieve neighbors of a node", "check if a neighbor exists", "get degree of a node", among other primitive graph operations used as building blocks. Most importantly, the method is efficient for big data and easily parallelized for both shared and distributed memory architectures. The method also has a variety of applications beyond the time and space improvements described above. For instance, one may use this technique to visualize graph data better as the large cliques can be combined to reveal other important structural properties [21, 22].

The compression is guaranteed to reduce space, IO costs, and improve performance of algorithms, as long as there exists cliques in the graph (of size $k \leq 3$). In large sparse graphs, the compression performs best for graphs that contain many reasonably sized cliques. Fortunately, many real-world networks are known to have such properties including social [23] and biological networks [24], web graphs, information networks, and many other real-world networks [6]. Notice that this strategy is suitable for directed networks as well, since cliques in these networks must be formed by reciprocal edges (edges in both directions, (u, v) and (v, u), as well as bicliques in bipartite graphs. As mentioned previously, the algorithm is effective as long as the size and frequency of the cliques are reasonable. The clique-based compression methods described in this work may be used as a succinct representation of the graph for both the following:

(1) In-memory graph encoding.
(2) Disk-resident graph encoding.

For both in-memory *and* disk-resident graph encoding, the methods have the following benefits:

- Reduce space/storage requirements.
- Speedup graph algorithms/computations and primitives.
- Reduce IO costs (e.g., CPU to GPU).
- Improve caching (and thus performance).

Summary of contributions

We propose a class of clique-based graph compression techniques. Our main contributions are as follows:

Query preserving compression

Given an arbitrary query Q on G, the query Q can be evaluated directly using the significantly smaller compressed graph G_c. Thus, any algorithm for answering Q in G can be applied to answer Q in the compressed graph G_c, without requiring decompression. This is in contrast to many existing techniques designed for social networks [25] and web graphs [26–29].

Support for arbitrary graph queries

The approach naturally generalizes for arbitrary graph queries including (i) graph matching queries that determine if a subgraph pattern exists or not, as well as (ii) reachability queries that determine whether a path exists between two vertices in G, among others.

Efficient incremental updates

Most real-world graphs are changing over time with the addition or deletion of nodes and edges [30, 31]. As opposed to existing methods, our approach provides efficient incremental maintenance of the compressed graph directly. Thus, the compressed graph is computed, and then incrementally maintained using fast parallel localized updates. In particular, given a new edge (v_i, v_j), our approach requires searching only the local neighborhood around that edge (to check if a larger clique of size $k+1$ is possible).

Parallel

The graph compression method is also scalable for massive data with billions or more edges. A parallel implementation of the algorithms is used and shown to be fast and scalable for massive networks.

Flexible

The method is flexible for use with other compression techniques. For instance, after using the clique-based compression, we can apply another standard compression technique to reduce the space requirements further. However, depending on the compression technique, it may only be useful for storing the graph, and not performing computations directly on the compressed graph data. This approach may benefit a wide range of applications such as relational learning [32], anomaly detection [33–35], role discovery [36], and community detection [37–39]. Furthermore, graph compression techniques have been used for various graph mining tasks including comparison of biological networks [40],

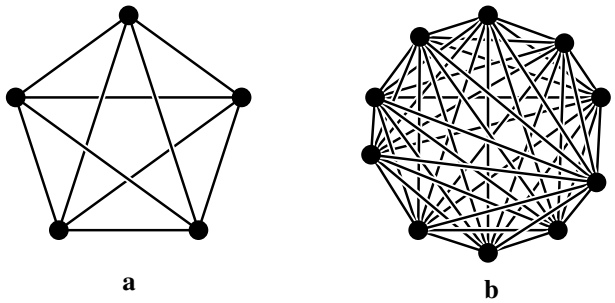

Fig. 1 A clique is a set of vertices any two of which are adjacent. The above examples shows cliques of size 5 and 10. **a** 5-clique. **b** 10-clique

frequent pattern discovery [41], summarizing graph streams [37], estimation of graph properties [42], among others [21, 43]. The methods are also useful for storing graphs with multiple edge types, etc. GraphZIP favors cliques of the largest possible size, as well as cliques that do not overlap with any cliques already detected by the method.

Methods

Preliminaries

Let $G = (V, E)$ be a graph. Given a vertex $v \in V$, let $N(v) = \{w \mid (v, w) \in E\}$ be the set of vertices adjacent to v in G. The degree d_v of $v \in V$ is the size of the neighborhood $|N(v)|$ of v. Let $diam(G) = max_{v,u} D(v, u)$ denote the diameter of G defined as the longest shortest path between any two vertices (v, u) of G where $D(v, u)$ is the graph distance between v and u, i.e., the minimum length of the paths connecting them. A clique is a set of vertices any two of which are adjacent. The maximum size of a clique in G is the clique number of G and is denoted by $\omega(G)$. We exploit many of the fundamental properties of cliques in the design of the parallel framework. Importantly, cliques are nested such that a clique C of size k contains a clique of size $k - 1$. Cliques are closed under exclusion, for instance, consider a clique C and $v \in C$, then $C - v$ is also a clique. Hence, a clique C is a heredity graph property, since every induced subgraph of C is also a clique. Finally, cliques can also be overlapping and in general may overlap in all but a single vertex. In network analysis and network science, cliques can be viewed as the most strict definition of a community. From that perspective, cliques are *perfectly dense* (degree of $k - 1$, larger degree not possible), *perfectly compact* ($diam(C) = 1$), and *perfectly connected*, i.e., C is a $k - 1$ vertex-connected and $k - 1$ edge-connected graph). Examples of cliques of size 5 and 10 are provided in Fig. 1. We also use the notion of k-cores. A k-core in G is a vertex induced subgraph where all vertices have degree of at least k. The k-core number of a vertex v denoted as $K(v)$ is the largest k such that v is in a k-core. Let K(G) be the largest core in G, then $K(G) + 1$ is an upper bound on the size of the maximum clique. Many of these facts are exploited in the design of the parallel framework for graph compression and encoding.

For each individual vertex, edge, and subgraph, we place an *upper bound* on the size of the maximum clique possible. These vertex, edge, and subgraph upper bound(s) enable us to potentially remove each individual vertex, edge, and subgraph for which a clique of maximum size cannot arise (using the individual upper bound on each graph element).

In-memory graph encoding

Now, we describe a generalization of the compressed sparse column/row (CSC/CSR) encoding [20] for the clique-based graph compression techniques.[1] This is particularly useful as an in-memory encoding of the graph. In particular, the generalization makes it easy for use them with many existing graph algorithms. To generalize CSC/CSR, a new vertex type is introduced called a *clique vertex*. Intuitively, the vertex represents a clique C in G and the size of the clique denoted $k = |C|$ must be of a sufficient size. The approach always finds and uses the largest such cliques in G to achieve the best compression. However, cliques of size $k > 2$ are beneficial as they may reduce space, IO costs, algorithm performance, and thus, this work requires cliques to be at least $k = 3$ and above. Obviously cliques of size 2 are not useful and would not improve compression. Clearly, the larger the cliques, the more compact the graph can be encoded, resulting in reduced IO costs, and so forth.[2]

The minimum id of a clique vertex (pseudo vertex) starts from $v_n + 1$, where v_n is the maximum id of any real vertex. That is,

$$V = \{\underbrace{v_1, v_2, \ldots, v_n}_{\text{normal vertices}}, \quad \underbrace{v_{n+1}, \ldots}_{\text{clique vertices}}\} \tag{1}$$

This way, as long as the algorithm knows v_n, it can quickly determine if a vertex is a clique vertex or not. In the latter case, the successor iterator would look up the definition of the clique and subsequently iterate over all the nodes in that clique for the correct successor generation.[3] If a neighbor of a normal vertex is a clique-vertex, then the iterator must retrieve the neighbors of the clique vertex. An example of a clique vertex is shown in Fig. 4c. For instance, v_{10} and v_{11} are clique vertices.

The above scheme includes the original CSC/CSR representation as a special case, that is, if no clique (and thus no pseudo vertex) exists in the graph. However, if one or more cliques exist, then the proposed clique compression technique is guaranteed to reduce the space needed to store the graph. Notice that thus far, the computational cost of various graph primitives (check if an edge exists,...) is essentially equivalent to the original CSC/CSR (see Fig. 2). For instance, iterating over the neighbors of a node requires us to scan the neighbors of a node just as before, with the addition that if a node with an id larger than v_n is scanned indicating a clique-vertex,[4] then we must also scan the neighbors of the clique-vertex. However, the proposed encoding is guaranteed to improve caching, since going to these pseudo-vertices and checking for a neighbor, etc. would occur frequently, and thus likely to be in cache, etc. Furthermore, in the most basic variants of the proposed techniques, cliques are vertex disjoint (since they are removed at same time they are discovered), thus, while a vertex may participates in many overlapping cliques, the method only finds a single large clique. After the graph compression/

[1] The CSR (CSC) encoding represents a sparse (adjacency) matrix **M** of a graph G by three arrays that contain nonzero values, the extents of rows (columns), and column (row) indices.

[2] However, since it requires additional time to find such cliques, then depending on the domain application it might be preferred to find cliques of a larger size. Thus, the methods allow for the user to adjust this parameter, which can improve runtime at the cost of space.

[3] A successor iterator is an enumerator that can be used to obtain a list of the neighbors of a node in some (usually deterministic) order.

[4] Note that clique vertices are at the end of the list of neighbors for a given vertex.

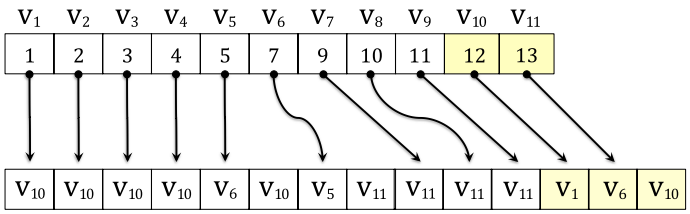

Fig. 2 Generalized CSR/CSC Data Structures (In-memory graph encoding). Cost of this particular in-memory graph encoding is 11 (or $\mathcal{O}(|V| + |\mathcal{C}|)$) for the ptr array (top) whereas the other is only 14, for a total of 27. This is compared to the popular CSC/CSR that requires arrays of size 9 and 34 for a total of 43. Thus, the simple encoding reduces the storage costs quite significantly (as well as IO costs, while improving algorithm performance as the computational costs of many graph primitives are also reduced using this encoding. Moreover, it is guaranteed to improve caching. Note that storing the graph to disk in a lossless fashion is even more efficient, since one can simply write "v_1 v_6 v_{10}" as the first line, followed by the edge v_5 v_6

encoding is complete, vertex ids are easily exchanged such that all vertices in a clique are assigned subsequent ids. Clearly, the benefit of exchanging the vertex ids in such a fashion is that many fundamental graph primitives are easily computed in $o(1)$ time. For instance, consider the primitive that checks if an edge exists (or checks if a node is a neighbor or not), then this takes only $o(1)$ time by simply checking the range (v_i, v_{i+k}) to determine if the node id is within that range or not. Hence, if we wanted to know if w is a neighbor of v, then we just need to go to that clique and check if w falls in that range. Clearly, this also improves caching.

Moreover, this also reduces space needed to store the graph (on disk), as we avoid writing each clique one by one, and can simply write the first vertex id of that clique. For instance, one possibility is to write the first vertex id of each clique (in order) on the first line of each file. See Fig. 3 for an example illustrating the space savings using such an approach to encode the graph in Fig. 4. To read this type of compressed graph from disk (assuming the graph reader simply reads the graph and encodes it using any arbitrary encoding), we would read the first actual line, and immediately know the cliques by simply examining the start id and the vertex start id of the next clique. Notice that this also reduces costly IO associated with reading the graph from disk, and/or the IO involved in sending the data from the CPU to GPU, or across the network in a distributed architecture, etc. Afterwards, the graph reader would proceed as usual reading the edges, etc.

The difference in cost between storing the cliques directly and the format that relabels vertices (that is, exchanges vertex ids) based on the cliques is $\mathcal{O}(\sum_{C \in \mathcal{C}} |C|)$ compared to $\mathcal{O}(|\mathcal{C}| + 1)$. As shown in "Results and discussion" section, this difference is significant for many real-world graphs since $|\mathcal{C}|$ is often thousands (and in many cases much larger), and the cliques in $\mathcal{C}$ can often be quite large as well. Cost of encoding all remaining edges is identical and thus not shown above. Note that the cost above is for disk-resident graph encodings. Though, the difference is similar for in-memory encodings.

Example

Given the simple graph in Fig. 4a, the method identifies the cliques of largest size shown in Fig. 4b. Figure 4c shows an intuitive example of a possible graph encoding scheme using the proposed framework. To store the graph compactly (e.g., to disk), notice that one can simply write the cliques to a file such that each line contains the vertices

$$\begin{bmatrix} 1 & 6 & 10 \\ 5 & 6 & \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \\ 2 & 3 \\ 2 & 4 \\ 2 & 5 \\ 3 & 4 \\ 3 & 5 \\ 4 & 5 \\ 5 & 6 \\ 6 & 7 \\ 6 & 8 \\ 6 & 9 \\ 7 & 8 \\ 7 & 9 \\ 8 & 9 \end{bmatrix}$$

Fig. 3 GraphZIP compressed format (left) vs. edge list without clique compression (right)

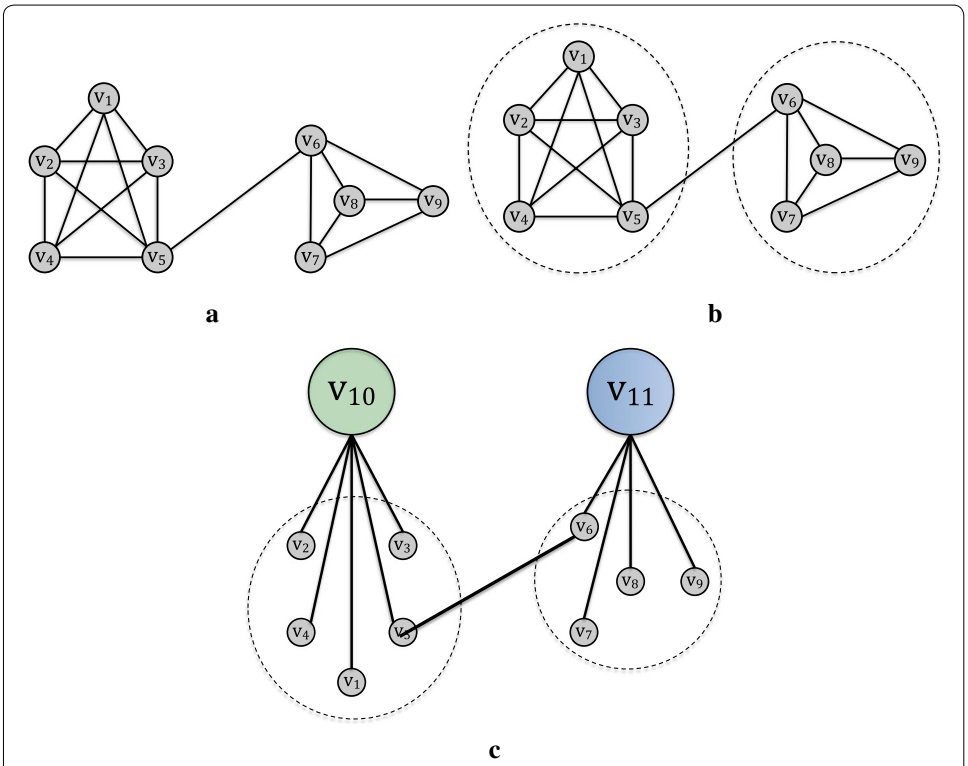

Fig. 4 Overview and intuition for the proposed graph compression techniques. **a** Simple graph. **b** Cliques identified. **c** Encoding Intuition

of a clique $C \in \mathcal{C}$, followed by the remaining edges (between nodes that do not share a clique). This can be reduced further by first recoding the vertex ids such that the vertex ids in the cliques are $C_1 = \{v_1, \ldots, v_k\}$, $C_2 = \{v_{k+1}, \ldots, \}$, and so on. Now, it is possible to encode the vertices in each clique by simply storing the first (min) vertex id. For instance, in-memory or on disk, the first vertex id of each clique is stored one after the other, for instance, storing v_1, v_6, v_{10} (in memory as an array, or to disk/a single line of a file) is enough since the clique C_i is retrieved by examining the ith position. That is, to retrieve the first clique in Fig. 4b, we retrieve the first vertex v_1 and can compute the last vertex id by the $i+1$ position, which corresponds to the first vertex id of the clique $i+1$. Thus, using this, we immediately get the last vertex id v_5. Furthermore, fundamental graph primitives (such as "checking if a node is a neighbor") may benefit from this encoding as many graph operations can be answered in $\wr(1)$ with a simple range check.

Algorithm

The set of cliques $\mathcal{C}$ (used in the compression schemes) is computed via the iterative algorithm in Algorithm 1. Once the set of cliques $\mathcal{C}$ are found (see Algorithm 1), any of the graph encoding methods may be used (e.g., see Fig. 4).

Heuristic and exact clique methods are used for finding the largest possible clique at each iteration (Algorithm 1 Line 4). The heuristic method is faster but results in slightly less compression, whereas the exact clique ordering takes longer, but results in a better compression. Many other variants also exist. For instance, one approach that performed well was to search only the top-k vertices (using either a heuristic or exact clique finder, that is, a method that for each vertex neighborhood subgraph $N(v)$ returns either a large clique or the maximum clique existing at that vertex neighborhood). The top-k vertices to search were chosen based on the maximum k-core number of each vertex.[5] This ordering can then be recomputed after each iteration as it is fast taking $\mathcal{O}(|E|)$ where $|E|$ is the number of edges remaining in G.

Algorithm 1 GraphZIP

```
1 input: G = (V, E) – a graph
2        δ – stopping criterion for smallest clique
3 repeat
4    Find a maximum clique C in parallel
5    Add C to the set of cliques C, that is, C ← C ∪ {C}
6    Remove C from the graph G
7 until all vertices are removed or |C| < δ
8 return the set of cliques C (for use with encoding schemes)
```

Many other variations of Algorithm 1 also exist. For instance, instead of removing C entirely (Line 6), one may allow some amount of overlap between the cliques in $\mathcal{C}$. Obviously, if there exists two cliques $C_i, C_j \in \mathcal{C}$ and both overlap significantly, then there is little benefit of using both in the encoding/compression.[6] However, allowing cliques that

[5] In general, any vertex ordering may be used.

[6] Significant overlap between cliques occurs frequently due to properties discussed in "Preliminaries" section.

overlap with only a few vertices may allow further reductions in space, at the expense of additional computational costs to find these cliques and ensure that there is not significant overlap. This can be formalized as,

$$\frac{|C_i \cap C_j|}{|C_i \cup C_j|} \geq \theta \tag{2}$$

where θ is the fraction of vertices allowed to overlap. Clearly, other types of "score functions" may also be useful and used interchangeably. As an aside, GraphZIP is also robust to missing data due to a property of cliques. For instance, if a node is missing at random, then the size of the clique for which this node would participate is at most decreased by one due to the hereditary property of cliques.

Parallel algorithm variant

Parallel variations of the proposed class of methods are discussed below. Let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be the set of independent sets. We obtain $\mathcal{S}$ using a greedy coloring method that attempts to find a *minimum* k and a mapping $s : V \to \{1, \ldots, k\}$ such that $s(v) = s(u)$ for each edge $(v, u) \in E$, see [44] for a large-scale study of greedy coloring of large complex networks. Select the independent set S_i from $\mathcal{S}$ with the largest set of vertices contained in the maximum k-core subgraph. Then find the largest clique for each vertex in S_i, and add them to the set of cliques $\mathcal{C}$. Since, we find cliques centered at the vertices in the independent set S_i, the cliques are guaranteed to be non-overlapping. This allows for the cliques to be found in parallel, without much effort. We choose the independent set containing the maximum number of vertices in the largest k-core since the maximum clique typically arises from these vertices. Order the vertices in the independent sets from the greedy coloring as follows:

$$\underbrace{S_1, S_2, S_3, \ldots, S_i, \ldots, S_k}_{\text{independent sets from greedy coloring}} \tag{3}$$

where the independent sets are ordered by binning the vertices in each S_i by k-core numbers, and selecting the set S_i with the maximum number of vertices in the largest possible k-core subgraph.

$$\underbrace{v_1, v_2, v_3, v_4, \ldots, v_i}_{S_1}, \underbrace{v_{i+1}, \ldots, v_j}_{S_2}, \cdots, \underbrace{v_h, v_{h+1}, \ldots, v_n}_{S_k} \tag{4}$$

This is computed in parallel by assigning each worker (i.e., processing unit) the next available vertex in the current independent set. Suppose there are $p_1, \ldots, p_t$ workers (i.e., processing units, cores) and the block size $\beta = 1$, then the first t vertices in S_1 in order are assigned to each of the t workers. The intuition behind the ordering is that we want to find the largest cliques first, and these are most likely in the largest k-cores.

Results and discussion

To evaluate the effectiveness of the proposed techniques, we use a wide variety of real-world graphs[7] from different application domains.

[7] Data accessible at http://www.networkrepository.com.

Table 1 GraphZIP compression results for a variety of real-world graphs. Note S_c denotes the space saved by GraphZIP defined as $S_c = 1 - \frac{G_c}{G}$

Graph	ρ	G (bytes)	G_c (bytes)	Space savings (S_c)	2χ comp. ratio	Time (s)
ia-enron-only	0.0614	4119	3004	27.07	69.94	0.006
ia-infect-dub.	0.0330	20,662	13,287	35.69	79.90	0.007
ia-email-univ	0.0085	41,868	32,405	22.60	79.34	0.017
bio-DD21	0.0009	136,847	85,510	37.51	80.72	0.139
ca-HepPh	0.0019	1,178,671	498,661	57.69	89.49	5.637
ca-AstroPh	0.0012	2,121,214	1,423,607	32.89	65.89	1.294
ca-CondMat	0.0004	1,001,147	695,000	30.58	80.46	0.892
ca-MathSciNet	0.0000	10,482,388	8,370,757	20.14	76.53	24.909
web-google	0.0033	23,043	12,216	46.99	81.42	0.004
web-BerkStan	0.0003	199,330	135,813	31.87	86.55	0.170
web-sk-2005	0.0000	4,083,783	2,040,783	50.03	92.60	9.030
web-indochina	0.0007	473,984	167,729	64.61	94.23	0.141
soc-gplus	0.0001	453,060	328,927	27.40	92.36	0.046
hamming6-2	0.9048	10,493	4916	53.15	90.59	0.004
hamming8-2	0.9686	226,317	110,838	51.03	96.32	0.745
c-fat200-1	0.0771	10,664	4674	56.17	84.57	0.003
chesapeake	0.2294	1035	749	27.63	58.94	0.001

Table 1 shows the effectiveness of GraphZIP for compressing a variety of real-world graphs. Note space savings is defined as the reduction in size relative to the uncompressed size: $S_c = 1 - \frac{G}{G_c}$. GraphZIP reduces the memory requirements by at least 20% across all graphs as shown in Table 1. Strikingly, we observe a 64% reduction in space when GraphZIP is used with web-indochina. Overall, GraphZIP appears to work best on web graphs followed by collaboration networks. Further analysis revealed that these graphs all have large cliques that are often much larger than cliques that arise in the other network types such as biological or interaction networks. We also report runtime performance in seconds. In most cases, GraphZIP takes only a few seconds as shown in the last column of Table 1. As an aside, the compression results in Table 1 are for the basic GraphZIP format where each clique $C \in \mathcal{C}$ is written on each line, followed by the remaining edges not in those cliques. Space can be further reduced using the GraphZIP format that relabels the vertex ids based on the cliques.

Results for compressing the encoded graph is shown in Table 2. These results use the same basic GraphZIP format used in Table 1 but reduce the space further by applying an additional compression technique to further compress the graph. This is most useful for storing large graphs on disk. Table 3 compares GraphZIP to the bvgraph compression method using Layered Label Propagation (LLP) order [45]. In both cases, GraphZIP compares favorably to this approach.

Figure 5 analyzes and compares the performance of four variants of GraphZIP. In particular, we compare a clique variant that uses an exact clique finder with degree ordering (exact-deg), exact clique finder with k-core ordering (exact-kcore), a heuristic clique finder with degree ordering (heu-deg), and a heuristic clique finder with k-core ordering (heu-kcore). The results indicate that performance is highly dependent on whether a heuristic or exact clique finder is used and the initial ordering (degree, k-core numbers),

Table 2 Results comparing the encoded graph G_c to the compressed $f(G_c)$

Graph	Space savings, $S_c = 1 - \frac{G_c}{G}$	
	Encoded graph, G_c	Compressed, $f(G_c)$
ia-enron-only	27.07	69.94
ia-infect-dub.	35.69	79.90
ia-email-univ	22.60	79.34
bio-DD21	37.51	80.72
ca-HepPh	57.69	89.49
ca-AstroPh	32.89	65.89
ca-CondMat	30.58	80.46
web-google	46.99	81.42
web-BerkStan	31.87	86.55
web-sk-2005	50.03	92.60
web-indochina	64.61	94.23
soc-gplus	27.40	92.36
hamming6-2	53.15	90.59
hamming8-2	51.03	96.32
c-fat200-1	56.17	84.57
chesapeake	27.63	58.94

Table 3 GraphzIP Compared to bvgraph compression with layered label propagation (Llp) order

Graph	$\|V\|$ (bytes)	$\|E\|$ (bytes)	G (bytes)	S_{LLC} (bytes)	S_C (bytes)	$S_{F(G_C)}$ (bytes)
Socfb-Penn	42K	1.4M	4237,507	40.49	46.56	78.16
Socfb-Texas	36K	1.6M	4,605,427	29.80	34.13	79.53

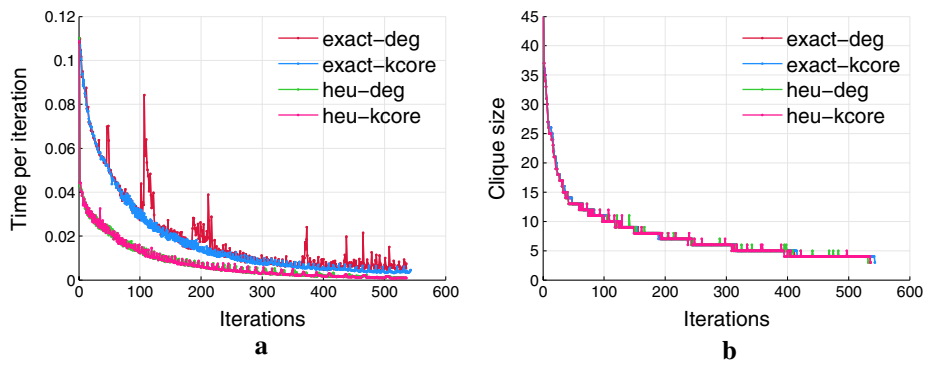

Fig. 5 Four variants of GraphZIP are compared. Runtime decreases with clique size as a result of the search space shrinking after each clique is found and removed from the graph. Further, the heuristic is 2-3 times faster than the exact clique finder. **a** Time. **b** Clique size

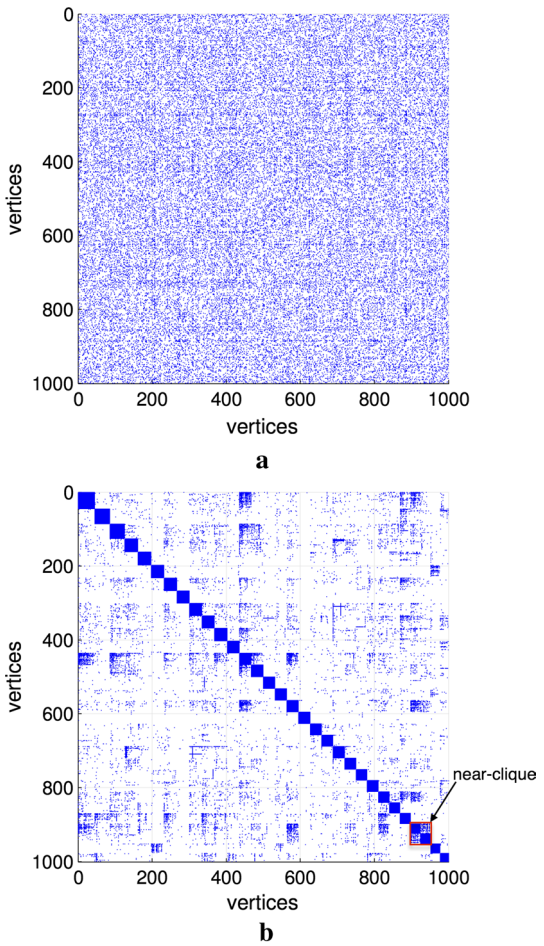

Fig. 6 Clique ordering is shown on the right compared to the initial ordering on the left. The dense off diagonal blocks reveal large cliques along the diagonal that are tightly connected and form much larger *near-cliques*. **a** Initial ordering. **b** Clique ordering

especially for the two exact variants (exact-deg, exact-kcore). Variants are straightforward to adapt to lower computational costs. Figure 6 shows the nonzero structure of the adjacency matrices using the initial vertex ordering compared to the clique ordering approach that orders vertices by the largest clique in which they appear. On the right, cliques are organized along the diagonal in descending order of size. The dense off diagonal blocks indicate other cliques that are tightly connected giving rise to a much larger *near-clique*

Conclusion

In this work, we described a class of of clique-based graph compression methods that leverage large cliques in real-world networks. Using this as a foundation, the proposed technique decomposes a graph into a set of large cliques, which is then used to compress and encode the graph succinctly. Disk-resident and in-memory graph encodings were proposed and shown to be effective with the following important benefits: (1) It reduces the space needed to store the graph on disk (or other permanent storage device) and

in-memory, (2) GraphZIP reduces IO traffic involved in using the graph, (3) it reduces the amount of work involved in running an algorithm on the graph, and (4) improves caching and performance. The graph compression techniques were shown to reduce memory (both on disk and in-memory) and speedup graph computations considerably. The methods also have a variety of applications beyond the time and space improvements described above. For instance, one may use this technique to visualize graph data better as the large cliques can be combined to reveal other important structural properties. Future work will explore using GraphZIP and its in-memory encoding to speedup common graph algorithms. This requires implementation of graph primitives that leverage the encoding. In addition, we will also investigate GraphZIP for compressing graphs from a variety of other domains.

Authors' contributions
All authors contributed equally in the conceptualization of GraphZIP and its design and development. Both authors read and approved the final manuscript.

Author details
[1] Adobe Research, 345 Park Ave, San Jose, CA, USA. [2] Google, 1600 Amphitheatre Pkwy, Mountain View, CA, USA.

Acknowledgements
We thank Dan Davies for discussions.

Competing interests
The authors declare that they have no competing interests.

Availability of data and materials
Data is available online at NetworkRepository [46]: http://networkrepository.com.

Ethics approval and consent to participate
Not applicable.

Funding
No funding to declare.

Publisher's Note

Received: 29 December 2017 Accepted: 24 February 2018
Published online: 03 March 2018

References

1. Fisher D, DeLine R, Czerwinski M, Drucker S. Interactions with big data analytics. Interactions. 2012;19(3):50–9.
2. Kambatla K, Kollias G, Kumar V, Grama A. Trends in big data analytics. J Parallel Distrib Comput. 2014;74(7):2561–73.
3. Herodotou H, Lim H, Luo G, Borisov N, Dong L, Cetin FB, Babu S. Starfish: A self-tuning system for big data analytics. CIDR. 2011;11:261–72.
4. Peshkin L. Structure induction by lossless graph compression. arXiv preprint arXiv:cs/0703132. 2007.
5. Gupta A, Verdú S. Nonlinear sparse-graph codes for lossy compression. IEEE Trans Inf Theory. 2009;55(5):1961–75.
6. Rossi RA, Gleich DF, Gebremedhin AH, Patwary M, Ali M. Parallel maximum clique algorithms with applications to network analysis and storage, vol 10. arXiv preprint arXiv:1302.6256. 2013.
7. Tomita E, Sutani Y, Higashi T, Takahashi S, Wakatsuki M. A simple and faster branch-and-bound algorithm for finding a maximum clique. In: WALCOM: Algorithms and computation. 2010. p. 191–203.
8. San Segundo P, Rodríguez-Losada D, Jiménez A. An exact bit-parallel algorithm for the maximum clique problem. Comput Oper Res. 2011;38:571–81.
9. Pullan WJ, Hoos HH. Dynamic local search for the maximum clique problem. JAIR. 2006;25:159–85.
10. Pattillo J, Youssef N, Butenko S. Clique relaxation models in social network analysis. Handbook of optimization in complex networks. Springer; 2012. p. 143–62.
11. Balasundaram B, Butenko S, Hicks I, Sachdeva S. Clique relaxations in social network analysis: the maximum *k*-plex problem. Oper Res. 2011;59(1):133–142.
12. Alba RD. A graph-theoretic definition of asociometric clique. J Math Soc. 1973;3(1):113–26.
13. Harish P, Narayanan P. Accelerating large graph algorithms on the GPU using cuda. In: HiPC. 2007. p. 197–208.
14. Vineet V, Narayanan P. Cuda cuts: fast graph cuts on the gpu. In: Computer vision and pattern recognition workshops (CVPRW). 2008. p. 1–8.
15. Zhou R. System and method for selecting useful smart kernels for general-purpose GPU computing. US Patent 20,150,324,707. 2015.

16. Liu X, Li M, Li S, Peng S, Liao X, Lu X. Imgpu: Gpu-accelerated influence maximization in large-scale social networks. IEEE Trans Parallel Distrib Syst. 2014;25(1):136–45.
17. Pan Y, Wang Y, Wu Y, Yang C, Owens JD. Multi-gpu graph analytics. arXiv preprint arXiv:1504.04804. 2015.
18. Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu WW. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: SIGPLAN. New York: ACM; 2008, p. 73–82.
19. Zhou R. Systems and methods for efficient sparse matrix representations with applications to sparse matrix-vector multiplication and PageRank on the GPU. 2015.
20. Kepner J, Gilbert J. Graph algorithms in the language of linear algebra. In: SIAM. 2011.
21. Von Landesberger T, Kuijper A, Schreck T, Kohlhammer J, van Wijk JJ, Fekete JD, Fellner DW. Visual analysis of large graphs: state-of-the-art and future research challenges. Computer Graphics ForumNew York: Wiley Online Library; 2011. p. 1719–49.
22. Ahmed NK, Rossi RA. Interactive visual graph analytics on the web. In: ICWSM. 2015, p. 566–9.
23. Traud AL, Mucha PJ, Porter MA. Social structure of facebook networks. Phys A Stat Mech Appl. 2011;391:4165–80.
24. Girvan M, Newman MEJ. Community structure in social and biological networks. In: PNAS. 2002;99(12):7821–6.
25. Chierichetti F, Kumar R, Lattanzi S, Mitzenmacher M, Panconesi A, Raghavan P. On compressing social networks. SIGKDD, 2009. p. 219–28.
26. Grabowski S, Bieniecki W. Tight and simple web graph compression. arXiv preprint arXiv:1006.0809. 2010.
27. Buehrer G, Chellapilla K. A scalable pattern mining approach to web graph compression with communities. In: WSDM. New York: ACM; 2008. p. 95–106.
28. Boldi P, Vigna S. The webgraph framework i: compression techniques. In: WWW. 2004. p. 595–602.
29. Suel T, Yuan J. Compressing the graph structure of the web. In: IEEE data compression conference. 2001. p. 213–22.
30. Kempe D, Kleinberg J, Kumar A. Connectivity and inference problems for temporal networks. STOC, 2000. p. 504–13.
31. Ahmed NK, Berchmans F, Neville J, Kompella R. Time-based sampling of social network activity graphs. In: SIGKDD MLG. 2010. p. 1–9.
32. Friedman N, Getoor L, Koller D, Pfeffer A. Learning probabilistic relational models. In: IJCAI. 1999. p. 1300–09.
33. Chandola V, Banerjee A, Kumar V. Anomaly detection: a survey. ACM Comput Surv. 2009;41(3):15.
34. Akoglu L, McGlohon M, Faloutsos C. Oddball: spotting anomalies in weighted graphs. In: Advances in knowledge discovery and data mining. 2010. p. 410–21.
35. Eberle W, Holder L. Anomaly detection in data represented as graphs. Intell Data Anal. 2007;11(6):663–89.
36. Rossi RA, Ahmed NK. Role discovery in networks. TKDE. 2015;27(4):1112–31.
37. Sun J, Faloutsos C, Papadimitriou S, Yu PS. Graphscope: parameter-free mining of large time-evolving graphs. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. New York: ACM; 2007. p. 687–96.
38. Prakash BA, Sridharan A, Seshadri M, Machiraju S, Faloutsos C. Eigenspokes: surprising patterns and scalable community chipping in large graphs. In: Advances in knowledge discovery and data mining. 2010. p. 435–48.
39. Leskovec J, Lang KJ, Dasgupta A, Mahoney MW. Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters. Internet Math. 2009;6(1):29–123.
40. Hayashida M, Akutsu T. Comparing biological networks via graph compression. BMC Syst Biol. 2010;4(Suppl 2):13.
41. Ketkar NS, Holder LB, Cook DJ. Subdue: compression-based frequent pattern discovery in graph data. In: Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations. New York: ACM; 2005. p. 71–6.
42. Ahmed NK, Duffield N, Neville J, Kompella R. Graph sample and hold: a framework for big-graph analytics. In: SIGKDD. New York: ACM; 2014. p. 1446–55.
43. Margaritis D, Faloutsos C, Thrun S. Netcube: a scalable tool for fast data mining and compression. VLDB, 2001.
44. Rossi RA, Ahmed NK. Coloring large complex networks. Soc Net Anal Mining. 2014;4(1):37.
45. Boldi P, Rosa M, Santini M, Vigna S. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. WWW. New York: ACM; 2011. p. 587–96.
46. Rossi RA, Ahmed NK. The network data repository with interactive graph analytics and visualization. In: Proceedings of the 29th AAAI conference on artificial intelligence. 2015. p. 4292–93. http://networkrepository.com