Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations
Towards Understanding of Modern Graph Processing and Storage

MACIEJ BESTA, Department of Computer Science, ETH Zurich
TORSTEN HOEFLER, Department of Computer Science, ETH Zurich

Various graphs such as web or social networks may contain up to trillions of edges. Compressing such datasets can accelerate graph processing by reducing the amount of I/O accesses and the pressure on the memory subsystem. Yet, selecting a proper compression method is challenging as there exist a plethora of techniques, algorithms, domains, and approaches in compressing graphs. To facilitate this, we present a survey and taxonomy on lossless graph compression that is the first, to the best of our knowledge, to exhaustively analyze this domain. Moreover, our survey does not only categorize existing schemes, but also explains key ideas, discusses formal underpinning in selected works, and describes the space of the existing compression schemes using three dimensions: areas of research (e.g., compressing web graphs), techniques (e.g., gap encoding), and features (e.g., whether or not a given scheme targets dynamic graphs). Our survey can be used as a guide to select the best lossless compression scheme in a given setting.

CCS Concepts: • Information systems → Data compression;

ACM Reference format:
Maciej Besta and Torsten Hoefler. 2018. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. 52 pages.

1 INTRODUCTION

Big graphs form the basis of many problems in machine learning, social network analysis, and various computational sciences [296]. Storage-efficient processing of such graphs is becoming increasingly important for HPC and Big Data. First, it may eliminate expensive I/O accesses. Moreover, it enables storing a larger fraction of data in caches, potentially increasing performance. Next, it could eliminate inter-node communication as graphs may fit in the memory of one node. Finally, reducing required amounts of memory lowers costs of necessary hardware.

There exists a plethora of graph compression schemes. They cover various fields such as web graphs [62], biology networks [125, 201], or social graphs [106]. Moreover, they follow different methodologies, for example attempting to build a graph representation that asymptotically matches the storage lower bound (so called succinct representations [155]) or permuting vertex integer labels to minimize the sum of differences between consecutive neighbors of each vertex and encoding these minimized differences with variable-length coding [61]. Next, there are different compression techniques that can be used in the context of any methodologies, for example reference
encoding [62], Huffman degree encoding [400], and many others. Finally, compression can be
general and target any graph [155] or may be designed for a particular class of graphs [55].

This paper aims to provide the first taxonomy and survey that attempts to cover all the associated
areas of lossless graph compression. Our goal is to (1) exhaustively describe related work, (2) illustrate
and explain the key ideas and the theoretical underpinning, and (3) systematically categorize existing
algorithms, schemes, techniques, methodologies, and concepts.

What Is The Scope of This Survey? We focus on lossless approaches and leave all the lossy
schemes for future work. The main reason for this is the fact that the scope of lossless graph
compression is on its own very extensive, covering almost 450 papers and various approaches,
numerous techniques, countless algorithms, and a large body of applications.

What Is the Scope of Existing Surveys? Existing surveys on graph compression cover only
a small part of the domain. Zhou provides a brief survey [440] with 14 references. Maneth and
Peternek [302] only cover a part of succinct representations, RDF graph compression, and a few
works categorized under the common name “Structural Approaches”. Finally, there are other
surveys that describe fields only partially related to graph compression: compressing polygonal
meshes [300], summarizing graphs [291], and compressing biological data [213]. We discuss these
surveys in more detail in the related parts of this work.

2 BACKGROUND
We first present concepts used in all the sections and summarize the key symbols in Table 1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G, A$</td>
<td>A graph $G = (V, E)$ and its adjacency matrix; $V$ and $E$ are sets of vertices and edges.</td>
</tr>
<tr>
<td>$n, m$</td>
<td>Numbers of vertices and edges in $G$; $</td>
</tr>
<tr>
<td>$d, \hat{d}, D$</td>
<td>Average degree, maximum degree, and the diameter of $G$, respectively.</td>
</tr>
<tr>
<td>$d_v, N_v$</td>
<td>The degree and the sequence of neighbors of a vertex $v$.</td>
</tr>
</tbody>
</table>
| $\Theta, \Theta_v, \Theta_v'$ | Data structures for, respectively, the pointers to the adjacency data of each vertex,
the adjacency data of a given graph $G$, and the adjacency data of vertex $v$. |
| $N_{in, v}, N_{out, v}$ | The in-neighbors and out-neighbors of a vertex $v$. |
| $N_{i,v}$ | The $i$th neighbor of $v$ (in the order of increasing labels). |

Table 1. The most important symbols used in the paper.

2.1 Graphs
We model an undirected graph $G$ as a tuple $(V, E)$ (denoted also as $G(V, E)$); $V$ is a set of vertices and
$E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. If $G$ is directed, we use the name arc to refer
to an edge with a specified direction. We consider both labeled and unlabeled graphs. If a graph is
labeled, $V = \{1, \ldots, n\}$, unless stated otherwise. $N_v$ and $d_v$ are the neighbors and the degree of a
vertex $v$. The $i$th neighbor of $v$ (in the order of increasing labels) is denoted as $N_{i,v}$; $N_{0,v} \equiv v$. We
use the name “label” or “ID” interchangeably. We denote the maximum degrees for a given $G$ as $\hat{d}$,
$\hat{d}_{in}$ (in-degree), and $\hat{d}_{out}$ (out-degree). $G$’s diameter is $D$.

We also consider more complex graph properties: arboricity and genus. Arboricity is a minimum
number of spanning forests that cover all the edges of a given graph. Next, a graph $G$ has a genus
$g \geq 0$ if it can be drawn without crossing itself (i.e., its edges) on the surface of a sphere that has $g$
handles (such as in a coffee mug). For example, a graph with $g = 1$ can be drawn on a torus [409].

2.2 Graph Representations
$G$ can be represented as an adjacency matrix (AM) or adjacency lists (AL). An AL consists of
a contiguous array with the adjacency data (denoted as $\Theta$) and a structure with offsets to the
neighbors of each vertex (denoted as $\Theta$). AL uses $O(n \log m + m \log n)$ bits while AM uses $O\left(n^2\right)$. 
We next describe families of graphs used in the following sections; we provide them to make our work self-contained. For simplicity, we focus on intuitive descriptions instead of formal definitions. General graphs can have multiple edges between the same two vertices (i.e., multigraphs) and loops (a loop is an edge starting and ending at the same vertex). Simple graphs have no loops and multiple edges between the same pair of vertices. Loop-free graphs are general graphs without loops. Stick-free graphs do not have vertices of degree one. A transposed graph has reversed arc directions of the input directed graph [61]. A graph embedding into a certain surface is a drawing of a graph $G$ on that surface so that $G$’s edges may intersect only at their endpoints [52]. A planar graph can be drawn on the plane so that its edges intersect only at their endpoints. An outerplanar graph has a planar drawing such that all vertices belong to the drawing of the outer face of the graph (intuitively, all the vertices lie on the boundary between the graph and the outer part of the “plane” surrounding the graph). A plane graph (also planar drawing or plane drawing) is a planar embedding of a planar graph. A $k$-page graph $G$ has its edges partitioned into $k$ sets $E_1, \ldots, E_k$ so that, for each $i$, a graph $G(V, E_i)$ is planar; $G(V, E_1)$ is a graph with a vertex set $V$ and edge set $E_1$ [237]. In a drawing of a graph $G$ on $k$ pages each vertex of $G$ constitutes a point on a “spine” of the book formed by these $k$ pages and each edge of $G$ is drawn as a curve within a single page. A $k$-page graph embedding on a given surface is a $k$-page drawing of a graph $G$ on this surface without any edge crossings. A $k$-connected graph $G$ (also called $k$-vertex connected) is such a graph in which one cannot find a set of $k – 1$ vertices whose removal disconnects $G$. A map is a topological equivalence class of planar embeddings of planar graphs. In encoding a map we are required to encode the topology of the embedding, i.e., incidences among faces, edges, and vertexes, as well as the graph. A map is an embedding of a unique graph, but a given graph may have more than one embedding. Thus, a map usually requires more bits to encode than a corresponding graph [243]. A plane triangulation is a plane graph, each of whose faces has size exactly three; a plane triangulation may contain self-loops and multiple edges. An Erdős–Rényi graph $G(n, p)$ [150] is a graph with a uniform degree distribution where $n$ is the number of vertices and $p$ is a probability that an arbitrary edge is present in the graph (independently of other edges). A separable graph is a graph in which we can divide $V$ into two subsets of vertices of approximately the same size so that the size of a vertex cut between these two subsets is asymptotically smaller than $|V|$. Other considered graph classes are graphs with bounded arboricity (intuitively, they are uniformly sparse) and graphs with bounded genus.

2.4 Codes

Finally, we summarize codes for encoding integers that are used in various works to encode adjacency arrays of vertices. Elias $\gamma$ [148] is a universal code for positive integers. It is often used when the maximum possible number to be encoded cannot be determined beforehand; when using this code, the size of a value $x$ is $2\lceil \log x \rceil + 1$ [bits]. Elias $\delta$ [148] is a universal and asymptotically optimal code for positive integers. With this code, the size of a number $x$ is $\lceil \log x \rceil + 2\lceil \log \lceil \log x \rceil + 1 \rceil + 1$ [bits]. Golomb [187] is an optimal non-universal prefix code for alphabets following a geometric distribution. The code is suitable for sequences of integers where small values are much more likely to occur than large values. Gray code [194] is an arrangement (i.e., a total ordering) of numbers (or other entities such as vectors) such that the binary representations of consecutive numbers or any other entities differ by exactly one bit. $\zeta$ [64] is a code suited for integers that
Fig. 1. (§ 3.1) The categorization of the considered domains of lossless graph compression.
follow the power law distribution with the exponent smaller than two. Finally, \( \pi \) \(^{[25]}\) is a universal code suited for integers that follow the power law distribution with the exponent close to one.

### 3 TAXONOMY AND DOMAIN DIMENSIONS

In this section, we describe how we categorize existing work in this survey. Figure 1 depicts the hierarchy of the considered domains.

#### 3.1 How Do We Categorize Existing Work?

Graph compression is related to various areas such as databases or information theory. Schemes in these areas often share various common features, for example addressing static or dynamic graphs. This poses a question on how to categorize the rich world of graph compression studies to enable its systematic analysis. In the following, we dedicate a separate section (§ 4–§ 8) to one particular area of research such as compressing graph databases. Thus, we devote one section to the work done within one specific community. Such community-driven areas constitute the first dimension of our categorization. We describe how we identify these areas in § 3.1.1. Second, different areas of graph compression may use the same techniques for compressing graphs, or various techniques may be used in one publication or algorithm. For example, it is common to combine gap encoding and variable-length codes to compress web graphs. The second dimension in our categorization are thus techniques that reduce the size of graphs. We dedicate § 3.1.2 to describe the most important techniques. Finally, applications of a given technique within a certain area may have different features. For example, one can use a specific technique for either static or dynamic graphs. Consequently, the third dimension are features; we describe them in § 3.1.3.

We also discuss the existing categorizations and taxonomies in § 3.2.

#### 3.1.1 Areas.

Many papers are dedicated to compressing graphs from specific domains such as web graphs, biological networks, social graphs, and others; we describe them in § 4. Second, we describe works related to compressing graph databases (§ 5). Next, various schemes that we list in § 6 are devoted to approaching the storage lower bounds while ensuring fast (ideally constant time) queries. Moreover, we discuss optimization approaches to improve graph layouts, which ultimately reduces space occupied by a graph. Finally, we devote a separate section for various works that cannot be categorized in one of the above (see § 8) and to areas related to graph compression and covered in other surveys (see § 9).

#### 3.1.2 Techniques.

We now briefly present several common techniques used in various areas, as well as examples of their usage, to improve the clarity of the survey.

**Variable-Length Encoding** In this technique, vertex IDs stored in the adjacency array are encoded with one of the selected variable-length codes such as Varint.

**Vertex Relabeling** The main idea is to change the initial IDs of vertices so that the new IDs, when stored, use less space. We also use the name vertex permutations to refer to this technique. This scheme is usually combined with variable-length encoding.
Relabeling combined with variable-length encoding reduces required storage

New labels are usually smaller than old ones
Variable-length encoding enables total size proportional to the size of vertex labels

Fig. 3. An example of vertex relabeling combined with variable-length encoding.

Reference Encoding Here, identical sequences of vertices in the adjacency arrays of different vertices are identified. Then, all such sequences (except for a selected one) are encoded with references [5, 367]. One can implement reference encoding with copy lists: sequences of 1s and 0s that indicate whether or not a given number is retained in the current adjacency array.

Run-length Encoding This scheme enhances copy lists in reference encoding. The key idea is to provide the size of consecutive sequences of 1s and 0s instead of the actual 1 and 0 values [28].

Huffman Degree Encoding The core idea in this scheme is to use fewer bits to encode vertex IDs of higher degrees. Thus, $|\mathcal{A}|$ is reduced as vertex IDs that occur more often use fewer bits.

Log Encoding This scheme uses $\lceil \log n \rceil$ bits to encode each vertex ID in a graph with $n$ vertices.

Interval Encoding Here, consecutive vertex IDs (e.g., $x, x+1, \ldots, x+k$) are stored using the interval boundaries $x$ and $x+k$.

Gap Encoding This scheme preserves differences between vertex IDs rather than the IDs themselves. The motivation is that, in most cases, differences occupy less space than IDs. Several variants can be used here; the most popular is storing differences between the IDs of the consecutive neighbors of each vertex $v$, for example $N_1(v) - v, N_2(v) - N_1(v), \ldots, N_{d_v-1}(v) - N_{d_v-2}(v), N_{d_v}(v) - N_{d_v-1}(v)$ (the first of the above differences is sometimes called an initial distance and each following: an increment). Assuming each $\mathcal{A}_v$ is sorted, one must use an additional bit to indicate the sign of the first difference. Another variant stores the differences between $v$ and each of its neighbors: $N_1(v) - v, N_2(v) - v, \ldots, N_{d_v-1}(v) - v, N_{d_v}(v) - v$.

There are many more techniques used in various areas of graph compression. We defer their description to the relevant parts of the survey. These are, among others, $k^2$-trees and their variants, hierarchical schemes based on the concept of supervertices and superedges, and schemes that reorder the rows or columns of the adjacency matrix of a graph. Some techniques are used in various areas but they are themselves actively developed and they constitute a separate area of graph compression. Examples are succinct data structures or schemes that relabel vertices.

3.1.3 Features. We briefly present several features used in various compression areas to improve the clarity of the survey. We later (§ 10) discuss selected features in more detail.

- Graph Dynamicity This feature indicates whether a graph to be compressed is assumed static or dynamic (and thus allowing for any changes to its structure or labels).
- Problem-Awareness This feature determines whether a given compression schemes is tuned to some specific algorithm or graph problem to be solved over the compressed graph.
- Graph-Awareness Here, we determine whether a given scheme is tuned (or designed) for some specific graph classes or whether it works for generic graphs.
- **Streaming Graphs** This feature indicates whether a given scheme addresses graphs that are processed as a stream of edges.

### 3.2 Existing Categorizations

We also survey the existing categorizations of graph compression schemes. First, Boldi et al. [61] indicate that permutations that relabel vertices can be *intrinsic* (also called *coordinate-free*) or *extrinsic*. The former relabel IDs basing only on the graph structure (i.e., vertices and edges). The latter also rely on some additional information such as URLs. Next, Dhulipala et al. [143] identify three categories of schemes for graph (and index) compression. First, there are *structural* approaches that find and merge repeating graph patterns (such as cliques). Second, there exist schemes for encoding adjacency data represented by a sequence of integers. Finally, various works propose vertex relabelings (label permutations) that minimize a given metrics, for example the sum of differences between consecutive neighbors in each adjacency list. Another study [335] distinguishes between *structural* approaches and the ones based on using the notion of entropy [387]. Finally, schemes for compressing web graphs use the notions of *locality* and *similarity* [58]. Locality means that most links from page $x$ point to pages on the same host (that often share a long path prefix with $x$). Similarity means that pages from the same host have many links in common.

### 4 COMPRESSING GRAPHS IN SPECIFIC DOMAINS

A large portion of research in graph compression is dedicated to compressing graphs in some specific domains. We now present the related efforts.

#### 4.1 Web Graphs

We start with web graphs.

1. **Initial Works.** The first works on web graph compression often use various combinations of techniques such as Huffman degree encoding, log encoding, gap encoding, differential encoding, and various variable-length codes. This domain was opened by the work on the Connectivity Server [53], a system for storing the linkage information found by the AltaVista [393] search engine. Connectivity Server associates each URL with an integer, sorts these integers (in each adjacency data structure $A_v$) according to the URL lexicographic order, and uses gap encoding on the integers.

   Another early work analyzes the web graph structure [81] and also enhances the original compression scheme in the Connectivity Server. Then, Wickremesinghe et al. [423] uses Huffman codes to encode references to links in the Connectivity Server. Moreover, Adler and Mitzenmacher [5] propose to compress web graphs with a Huffman-based scheme applied to in-degrees. They also use reference encoding and log encoding.

   Suel and Yuan [400] compress URLs using common prefixes. To compress links, they distinguish between global links connecting sites on different hosts and local links connecting sites on the same host. For global links, they first identify $p$ URLs with the highest in-degree, and encode links to these URLs with a Huffman code. For other global links, they use log encoding or encode the link with a Golomb code, depending on the out-degree of a given URL. Next, they identify two local link classes. For each host $h$, they determine the given number of most popular destinations for local links within $h$. Links to these pages are encoded with a Huffman code.

   The Link Database [367] is a system for storing web graphs. It uses various techniques (delta codes, Huffman codes, Grey orderings, and nybble codes); the related analysis identifies two important web graph properties, namely *locality* and *similarity*. The Link Database compresses offset arrays $O$ by using different bit counts to store offsets for different degree ranges. For example, it uses
a 32-bit index to the start of a given range of vertices, and then only 8-bit offsets for each of the following vertices.

4.1.2 Text-Related Works. Next, we present efforts that revolve around treating the input graph $G$ as text and using the associated compression methods. Navarro [328] proposes to regard $G$ as text and to utilize existing techniques for text compression and indexing. Specifically, he uses the Compressed Suffix Array (CSA) [376] structure as a basis for his graph representation. CSA is a compressed full-text self-index: a data structure built over a text $T = t_1...t_n$ (over an alphabet $\Sigma$) that requires space proportional to the size of the compressed text and simultaneously enables accessing any substring of $T$ (including the whole $T$ itself). Thus, it becomes unnecessary to store $T$ and some search operations on it are enabled. Now, the key idea is to treat an input graph as $T$ and use various CSA's functionalities to enable accessing the graph efficiently without decompressing it.

Claude and Navarro [117] use Re-Pair [270]: a phrase-based compressor that permits fast and local (i.e., without having to access the whole graph) decompression. Re-Pair repeatedly finds the most frequent pairs of symbols in a sequence of integers and replaces them with new symbols, until no more replacements reduce storage. An example is in Figure 5.

The approach based on Re-Pair was further extended by Claude and Navarro [118]. They combine it with an approach based on perceiving $G$ as a binary relation on $V \times V$ and use several techniques developed specifically for answering queries over binary relations [40, 41]. The motivation for such a combination is to derive both $N_{out,v}$ and $N_{in,v}$ fast.

4.1.3 $k^2$ Trees. We now describe efforts related to so called $k^2$ trees. In their seminal work, Brisaboa et al. [78, 188] present a Web graph representation that uses a tree structure that takes advantage of the structure of the adjacency matrix $A$ of web graphs. Specifically, it uses the sparseness and clustering properties of $A$. An example is presented in Figure 6. Initially, $A$ is divided into $k^2$ submatrices of the same size ($k$ is a parameter); these submatrices are recursively divided in the same way. Now, the key idea is to represent $A$ as a $k^2$-ary tree (called a $k^2$ tree) that corresponds to the above recursive “partitioning” of $A$. Each tree node contains a single bit of data. Each internal tree node has $k^2$ children. At each partitioning level, if a given submatrix to be partitioned contains only 0s, the corresponding tree node contains 0. Otherwise, it contains a 1. The resulting tree is encoded using a special simplified tree encoding that ensures asymptotically low compression ratio [365]. The $k^2$ representation ensures obtaining $N_{out,v}$ and $N_{in,v}$ fast; it also enables extended functionality not usually considered in compressed graph representations.

Claude and Ladra [116] further extended the $k^2$ tree idea by combining it with the Re-Pair scheme. Specifically, they first split the graph into subgraphs that correspond to different web domains. After that, the key idea is to encode each subgraph with a $k^2$ tree representation and encode the remaining inter-subgraph edges with Re-Pair.

A study on the best data distribution for query processing on graphs compressed with $k^2$ trees can be found in the work by Alvarez et al. [21].
An adjacency matrix of some graph:

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
numerical values that identify (and point to) each submatrix; lower numbers are assigned to more frequent submatrices to lower the total number of bits used to represent the matrices.

4.1.4 WebGraph Framework. We devote a separate subsection to describe the efforts related to the WebGraph Framework, a mature framework targeted at compressing web graphs.

Boldi and Vigna provide several works on compressing Web graphs. First, they developed the WebGraph framework [62] that is a freely available suite of codes, algorithms, and tools for compressing graphs with a special focus on web graphs. In addition to exploiting the two well-known web graph properties (locality and similarity) they also identify three further properties: (1) “similarity concentration”, i.e., either two adjacency lists have almost nothing in common, or they are characterized by large overlap, (2) “consecutivity is common”, i.e., many links within a page are consecutive with respect to the lexicographic order, and (3) “consecutivity is the dual of distance-one similarity”, i.e., if there is a lot of similarity between vertices and their respective successor lists in lexicographical ordering, the transpose of the given graph must contain large intervals of consecutive links. Now, WebGraph uses the following compression techniques: (1) gap encoding, (2) reference encoding, (3) differential encoding, and (4) interval encoding. Reference encoding may have an arbitrary number of levels, bounded by a user parameter. Another parameter controls the minimum number of integers that enables a given sequence of numbers to be taken into consideration when applying reference encoding. Finally, for high performance, adjacency lists are build lazily (i.e., the data available through reference encoding is only fetched when required).

Moreover, Boldi et al. [60, 61] test several existing vertex permutations. They also propose two new permutations based on the Gray [258] ordering. The key idea is to permute the rows of the adjacency matrix so that the adjacent rows change according to the Grey code. Moreover, they analyze transposed graphs and illustrate that coordinate-free permutations improve compression rates (coordinate-free permutations are permutations that achieve almost the same compression performances regardless of the initial ordering). They conclude that the Grey ordering may improve the compression rates and that coordinate-free orderings are particularly efficient for transposed graphs (in some cases reaching the level of 1 bit per edge).

Boldi et al. [57] conduct an analysis that aims to formally understand why the existing approaches compress web graphs well. For this, they first observe that it is important for high compression ratios to use an ordering of vertex IDs such that the vertices from the same host are close to one another. To understand this notion more formally, they propose measures of how well a given vertex ordering \( \pi \) respects the partitioning \( \mathcal{H} \) induced by the hosts. Their first measure is the probability to experience a host transition (HT), i.e., a fraction of vertices that are followed (in the \( \pi \) ordering) by another vertex at a different host:

\[
HT(\mathcal{H}, \pi) = \frac{1}{n} \cdot \sum_{i=1}^{n-1} \delta \left( \mathcal{H} \left[ \pi^{-1}(i) \right], \mathcal{H} \left[ \pi^{-1}(i-1) \right] \right)
\]

where \( \delta \) is the usual Kronecker’s delta and \( \mathcal{H}[x] \) denotes the equivalence class of a vertex \( x \): the set of vertices with the same host as \( x \).

Moreover, the second used measure is adapted from work by Meilà [315] and is called the Variation of Information (VI); it enables comparing two partitions \( \mathcal{H} \) and \( \mathcal{H}_\pi \) that are associated with the original vertex ordering and the \( \pi \) ordering. First, VI uses a notion of entropy \( H(\mathcal{P}) \) associated with a host partitioning \( \mathcal{P} \):

\[
H(\mathcal{P}) = - \sum_{S \in \mathcal{P}} P(S) \log(P(S))
\]

where \( P(S) = \frac{|S|}{n} \). Then, the mutual information between two partitions is defined as...
\[
I(\mathcal{P}, \mathcal{T}) = \sum_{S \in \mathcal{P}} \sum_{T \in \mathcal{T}} P(S, T) \log \frac{P(S, T)}{P(S)P(T)}
\]

(3)

where \(P(S, T) = \frac{|S \cap T|}{n}\). Finally, the VI measure is then defined as

\[
VI(\mathcal{P}, \mathcal{T}) = H(\mathcal{P}) + H(\mathcal{T}) - 2I(\mathcal{P}, \mathcal{T})
\]

(4)

Now, substituting \(\mathcal{P}\) and \(\mathcal{T}\) with \(\mathcal{H}\) and \(\mathcal{H}_{\pi}\) and observing that \(I(\mathcal{H}, \mathcal{H}_{\pi}) = H(\mathcal{H})\) gives

\[
VI(\mathcal{H}, \mathcal{H}_{\pi}) = H(\mathcal{H}_{\pi}) - H(\mathcal{H})
\]

(5)

the authors use the HT and VI measures to compare various vertex orderings used for web graphs. In the second part of the paper, they show that their ordering called Layered Label Propagation (explained in more detail in § 4.2 dedicated to social networks as it was introduced mostly in the context of social networks) outperforms other orderings proposed in the literature.

Other works due to Boldi, Vigna, and others include developing complete instantaneous \(\zeta\) codes for integers distributed as a power law with the exponent smaller than two [64], and presenting how to implement WebGraph with Java [63]. Some benchmarks about Web Graph can also be found in several empirical analyses [181, 271].

4.1.5 Hierarchical Schemes. We now survey methods where a given part of the input graph (e.g., a clique or a subgraph) is collapsed into a smaller entity (e.g., a vertex) to ultimately reduce space.

Raghavan and Garcia-Molina [363] propose a representation called S-Node. They find and collapse subgraphs into supervertices and then use Huffman encoding on in-degrees as well as reference encoding. Edges between supervertices are merged and form superedges, with information on which single edges should be added or removed to restore the original graph structure. To further enhance the representation, they incorporate various types of graph partitioning and they utilize contiguous vertex IDs with respect to their orderings in supervertices.

Buehrer and Chellapilla [84] generate virtual nodes from frequent itemsets in the adjacency data (i.e., dense subgraphs are replaced with sparse ones). For example, consider a fully connected directed bipartite subgraph where \(n_1\) and \(n_2\) denote the numbers of vertices in each of the two color classes; all the edges are directed towards one color class. The proposed scheme collapses the edges in such a subgraph by introducing a virtual node \(v\). Now, \(n_1\) edges from one class point to \(v\) and \(n_2\) edges point from \(v\) to each vertex in the other color class. This reduces the edge count from \(n_1 n_2\) to \(n_1 + n_2\) and is able to achieve a high compression ratio as bipartite subcliques are frequent in web graphs [264]. In this work, the complexity of the mining phase (i.e., where itemsets are found) is bounded to \(O(m \log m)\). To ensure this complexity they first cluster similar vertices in the graph (vertices are similar if a significant portion of their outlinks point to the same neighbors). Next, they find patterns in the clusters, remove the patterns, and replace these patterns with virtual nodes. Buehrer and Chellapilla’s work was extended by Mondal [320] by providing insights into which values of respective algorithm parameters are best suited in a given scenario.

Karande et al. [241] complement the work of Buehrer and Chellapilla [84] by showing that various web graph algorithms can be extended to run on graphs compressed with virtual nodes so that their running times depend on the size of the compressed graph instead of the original \(G\). They consider algorithms for link analysis, estimating the size of vertex neighborhoods, and various schemes based on matrix–vector products and random walks (PageRank [73], HITS [255], and SALSA [274]). For example, they show how to transform PageRank results for a compressed graph to obtain the PageRank results for the original graph.
Khalili et al. [245] relabel vertices so that similar vertices have closer IDs. Second, they group similar vertices together and collapse edges between groups into single superedges. To keep track of the collapsed edges they use an auxiliary data structure.

Anh and Moffat propose a hierarchical scheme for compressing web graphs [24]. The key idea is to partition the adjacency arrays into groups of \( h \) consecutive arrays. Then, sequences of consecutive integers in each of the \( h \) arrays are replaced with new symbols to reduce \( |\mathcal{A}| \); this can be seen as grammar-based compression conducted in a local manner [192].

Grabowski and Bieniecki [191, 192] present two schemes; the first one offers higher compression ratios while the second one is faster. The first scheme extends the reference encoding from the WebGraph framework. Among others, it extends the binary format of reference encoding by using more than two values to indicate more possible referenced values. Another one creates blocks of \( h \) adjacency lists. Then, within each such block, \( h \) adjacency lists are merged and then all duplicate numbers in each merged sequence of lists are removed. Simultaneously, every number receives an associated list that indicates which neighborhoods it originally belongs to.

Hernandez and Navarro [206] combine several techniques to accelerate obtaining various graph properties and to further reduce the required storage. First, they combine reducing the number of graph edges (through virtual nodes [84]) with vertex reordering [25, 61, 62] to better exploit ordering, locality, similarity, and interval/integer encoding. Next, they combine \( k^2 \) trees [78] with virtual nodes [84] as well as several other techniques, including Re-Pair [117]. They address both web and social graphs. They first analyze various existing compression methods for web graphs and show that a combination of schemes that collapse edges and reorder vertices can offer outstanding compression ratios and performance of graph accesses. Second, they propose a novel compression method that uses compact (as defined in § 6) data structures to represent social communities.

In other studies, Hernandez and Navarro [207, 208] propose to find dense subgraphs and represent \( G \) as the set of dense subgraphs \( \mathcal{H} \) plus a remaining graph \( R \). For this, they first investigate the notion of dense subgraphs and define dense subgraphs to be pairs \((S, C)\) of subsets of vertices, such that every vertex in \( S \) points to every vertex in \( C \), but with \( S \) and \( C \) not necessarily disjoint. Thus, the case where \( S = C \) corresponds to cliques and the case where \( S \cap C = \emptyset \) corresponds to bicliques. The authors show that subgraphs where \( S \neq C \) (without being disjoint) occur often in web (and social) graphs and designing a compressed representation for them pays off. Second, they store \( \mathcal{H} \) using a combination of integer sequences and bitmaps. To store \( R \), they use several existing techniques such as \( k^2 \) trees [78, 79, 268], WebGraph schemes with the Layered Label Propagation ordering [57, 62], and \( k^2 \) partitioning [116].

Maneth and Peternek [303, 304] recently proposed a scheme that recursively detects repeated substructures and represents them using grammar rules. They show that some queries such as reachability between two nodes can be evaluated in linear time over the grammar, enabling speedups proportional to the compression ratio. The key idea, similarly to Claude and Navarro [117], is to use Re-Pair [270]: a phrase-based compressor that permits fast and local decompression. However, contrarily to Navarro’s work, they do not use Re-Pair over a string built from the adjacency list, but instead represent frequent substructures with grammar rules.

4.1.6 Others. Guillaume et al. [197] aim to efficiently encode large sets of URLs using only widely available tools, namely gzip and bzip [141]. They also aim to make the mapping between URLs and their identifiers as fast as possible, and to compute \( N_{out,v} \) efficiently. To avoid decompressing the entire list of URLs, they split the sequence to be compressed into blocks and compress each block independently. They utilize gap encoding, focusing on differences between a given vertex and each of its neighbors to derive the length of each link. They also observe that such link lengths
follow a power distribution. Each length of a link is represented in either a Huffman code, 16-bit integer, or 32-bit integer according to its absolute value.

Asano et al. [28] encode integers which have a power distribution with a generalization of the variable-length nybble code. They use Kraft’s inequality [262] about instantaneous codes to show that, when a random variable $X$ has a probability function $f(X)$, the instantaneous code which minimizes average codeword length when used to represent $X$ is $\log f(x)$ bits long when encoding $x$. Thus, if $X$ follows the power distribution with the exponent $-\alpha$, the instantaneous code minimizing the average codeword length is the variable-length block code with $\frac{1}{\alpha-1}$-bit blocks. Next, they show that, when each $A_v$ is gap encoded, the first numbers in each $A_v$ and the accompanying increments follow power distributions of different exponents. They use this to develop a new encoding of the web graph. Consider the $A_v$ of any $v$. Suppose $v$ has out-degree $d$. Then $A_v$ has one initial distance and $d-1$ increments. Consecutive 1s in the list of increments are compressed using the run-length encoding [359]. Finally, the initial distance, the increments, and the run-length codes are represented in the variable-length block codes with 6-bit, 3-bit, and 1-bit blocks, respectively.

The main idea due to Asano et al. [29] is to identify identical blocks in the adjacency matrix $A$ and then represent $A$ with a sequence of blocks combined with some metadata information on the block type and others. Now, they propose to use six different types of such blocks that correspond to different types of locality within each host in the input web graph. For a thorough analysis, they provide a detailed classification and an extensive discussion on the proposed locality (and thus block) types. Inter-host links are treated as related to a special type of locality.

Apostolico and Drovandi target both web graphs and more general graphs [25]. Instead of naming vertices based on the lexicographical ordering of their URLs (and thus being tailored for web graphs only), they conduct a breadth-first search traversal of the input graph and assign an integer to each vertex according to the order in which it is visited. This ensures significant storage reductions after gap encoding is applied. They also introduce a new class of $\pi$-codes: universal codes for integers that follow power law distribution with an exponent close to one.

Dhulipala et al. [143] extend the work due to Chierichetti et al. [106] and show how it can be employed for compression-friendly vertex relabeling in social networks and web graphs. They first note that optimal compression-friendly relabeling of vertices is NP-hard. Their key idea is to reduce the size of the problem domain. For this, they recursively bisect the graph and, once the size of the partitions is small enough, compute a selected (possibly optimal) reordering for each partition. Finally, these partial results are combined to obtain the solution for the whole graph.

Analysis of the impact of various coding schemes on the compressibility of link graphs was done by Hannah et al. [199]. Breyer [72] presents the MarkovPR software that optimizes storing URLs in web graphs with a large trie and hashtables that alleviate navigating in the trie. Finally, W-tree [33] is a space-efficient representation for web graphs optimized for external memory settings.

### 4.2 Social Networks

Several recent works aim to specifically condense social networks. Some offer novel schemes while others investigate how to reuse the schemes developed for web graphs.

Chierichetti et al. [106] provide three contributions targeted at social network compression. First, they prove hardness results about several types of vertex reordering; we provide more details in a section devoted to vertex relabeling (§ 7). Second, they propose the BL compression scheme that extends Boldi and Vigna’s BV scheme from the WebGraph framework. BL takes advantage of a certain property common in social networks, namely reciprocity, next to the properties of locality and similarity. Reciprocity means that most unidirectional links are reciprocal, i.e., there is a link with a reverse direction connecting the same vertices. Third, the final design contribution
is the shingle ordering that preserves both locality and similarity. Intuitively, it treats $N_{out,v}$ as a set and derives a special value called the shingle $M_\sigma(N_{out,v})$ of this set where $\sigma$ is a suitably selected permutation (or hash function). Then, the vertices of the input graph are ordered by their shingles. The authors show that, if two vertices have many outneighbors in common, then with high probability they will have an identical shingle and they will be close in the shingle ordering.

Maserrat and Pei [311, 312] aim to answer both $N_{in,v}$ and $N_{out,v}$ in sublinear time in $n$ and $m$ while compressing the input graph. For this, they propose an Eulerian data structure: a structure that stores a linearization of the input graph in a space-efficient way and uses it to answer the neighborhood queries efficiently.

Boldi et al. [57] propose Layered Label Propagation (LLP), a compression-friendly vertex ordering targeting social networks. They start their work with an analysis that aims to formally understand why the existing approaches compress web graphs well (see § 4.1.6 for a detailed discussion on this part of their work). Still, the bulk of the paper is dedicated to the LLP ordering that targets social networks in the first place. To understand LLP, we first explain three other related schemes: a generic label propagation algorithm, a simple label propagation algorithm (LPA) [364] and a variant of the Absolute Potts Model (APM) scheme [373] that builds upon LPA.

First, any label propagation algorithm executes in rounds. At each round, every vertex updates its label according to some rule; this rule’s exact design constitutes the difference between various label propagation algorithms. Before the first iteration each vertex has a different label; the algorithm terminates when no more update takes place.

Second, in LPA, a vertex decides to adopt a label that is used by most of its neighbors. Its main problem is that it tends to produce one giant cluster with the majority of vertices.

APM addresses this issue. Assume a vertex $v$ has $k$ neighbors and let $\lambda_1, \ldots, \lambda_k$ be the labels belonging to $v$’s neighbors. Let also $k_l$ and $v_l$ be the number of $v$’s neighbors with a label $\lambda_l$ and the total number of vertices in $G$ with $\lambda_l$, respectively. Now, when updating its label, instead of selecting a label $\lambda_l$ that has the maximum value of $k_l$, $v$ selects a label that maximizes the value $k_l - \gamma(v_l - k_l)$. Intuitively, this rule does not only increase the density (i.e., the number of edges) of a given community (which happens because $k_l$ new edges adjacent to $v$ join a given community), but also decreases it because of $v_l - k_l$ non-existing edges. The $\gamma$ parameter controls the importance of each of these two effects. This strategy prevents generating one huge giant cluster.

Now, to understand the idea behind the LLP scheme, first observe that different values of $\gamma$ unveil clusters of different resolutions. If $\gamma$ is close to 0 it highlights large clusters (when $\gamma = 0$ then APM degenerates to LPA); increasing $\gamma$ unveils small clusters. LLP attempts to obtain a labeling that considers clusters of various resolutions. In general, it iteratively executes APM with various $\gamma$ values. Now, each such iteration outputs a vertex labeling. Vertices within the same cluster maintain the same order from past iteration. Vertices that acquired the same label are attempted to be placed as close to one another as possible.

Boldi and Santini [58] show in more detail advantages of using a clustering algorithm described in the LLP paper [57] for web graphs. They discuss how to use it to enhance both locality and similarity and provide several interesting examples visualized with associated adjacency matrices.

Shi et al [390] illustrate that the $k^2$ tree representation (§ 4.1.3) can be enhanced in several ways. Among others, they propose to use the DFS vertex order combined with a heuristic that reorders the adjacency matrix to make sure the cells with “1” are concentrated in few submatrices. To achieve this, the heuristic uses the Jaccard coefficient for the structural similarity of any two vertices.

Liakos et al. [280, 281] use the fact that the LLP reordering enhances the locality in such a way that the corresponding AM contains a large “stripe” around its diagonal that groups a large fraction
of edges. They use a bitvector to represent these edges and ultimately reduce space to store a network. Finally, Cohen briefly discusses various strategies for social network compression [121].

4.2.1 Combining Web and Social Networks. Among the works described in § 4.1 and § 4.2, some are dedicated to compressing both web and social networks [58, 60, 106, 116, 143, 206–208, 312]. Zhang et al. [438] propose the bound-triangulation algorithm. The main idea is to use a data structure that stores triangles efficiently. The motivation is that many web graphs and social networks contain a large number of triangles, thus priority placed over storing this motif efficiently reduces the required storage. Angelino [23] proposes a new vertex ordering that considers semantic data associated with the graph. For example, they propose to sort vertex neighborhoods by a selected property such as “name”. Miao [316] extracts dense subgraphs from web and social graphs and encodes them using succinct data structures such as wavelet trees.

4.3 Biological Networks
A significant amount of work is dedicated to compressing biological networks. The vast majority are related to genome assembly networks [388]. Besides that, few others exist, for example on compressing gene regulatory networks [145] and metabolic graphs [34], or optimizing protein network alignment [233]. There also exists a survey [213] on compressing various types of biological data (not necessarily graphs).

4.3.1 Schemes Based on De Bruijn Graphs. De Bruijn graph [133, 189, 380] is a directed graph that represents overlaps between sequences of symbols. For a given set of symbols of cardinality $s$, the corresponding $N$-dimensional De Bruijn graph has $s^N$ vertices consisting of all possible sequences of these symbols; each symbol may appear more than once in a sequence. An edge from a vertex $v_1$ to a vertex $v_2$ exists iff we can shift all the symbols associated with $v_1$ by one place to the left and add a symbol at the end, and ultimately obtain the sequence associated with $v_2$.

De Bruijn graphs are commonly used in the de novo genome assembly [37, 88, 124, 127, 171, 278, 347, 397, 434], which is one of fundamental bioinformatics projects. Some specific applications include assembly of DNA sequences [216], mRNA [190] assembly, metagenome assembly [348], genomic variants detection [217, 351] and de novo alternative splicing calling [375].

Genome assembly process builds long contiguous DNA sequences (called contigs) from a set of much shorter DNA fragments (called reads). Assemblers based on De Bruijn graphs first extract subsequences (mers) of length $K$ from reads ($K$ is a parameter). Then, a De Bruijn graph consisting of mers as vertices is built and then simplified, if possible. Now, contigs are simple paths in this graph and then can be extracted by finding a Hamiltonian or (more preferably) an Eulerian path.

Construction and navigation of the graph is a practical space and time bottleneck, which is why space-efficient representations of de Bruijn graphs have been researched intensely. The storage lower bound ($\text{[bits]}$) of a De Bruijn graph constructed from $K$-mers is $\log_2 \binom{4^{K+1}}{m} (m = |E|)$.

Li et al. [278] were the first to use De Bruijn graphs in assembly of human genome with mers large enough to detect structural variation between human individuals, to annotate genes, and to analyze genomes of novel species. They used minimum-information de Bruijn graphs without the information on read locations and paired-end information. There were various previous short-read assemblers, including EULER [353], Velvet [435], ALLPATHS [88], and EULER-SR [96]. Yet, they all are targeted at bacteria- or fungi-sized genomes, and are mostly unable to manage large genomes. ABYSS is another assembler, implemented with MPI on distributed-memory machines for more performance [396]. It avoids using pointers in a De Bruijn graph representation for memory savings; the graph is represented as a distributed hash table, acting as a mapping from a $K$-mer to a byte with the connectivity information related to this mer.
Ye et al. [430] show how to construct a graph equivalent to the de Bruijn graph by storing only one out of $d$ vertices ($d \in [10; 25]$). Their approach involves skipping a fraction of $K$-mers to reduce memory consumption. As an example, assume that there are two pairs of overlapping vertices: $A, B$ and $B, C$. The authors simply store the overlap $(A, C)$ instead of two overlaps $(A, B)$ and $(B, C)$, eliminating read $B$ from the graph. They attempt to sample one out of every $g (g < K)$ $K$-mers.

Cazaux et al. [93], Minkin et al. [318], and others [48, 179, 283, 369, 374] show fast and space-efficient algorithms for constructing compact De Bruijn graphs.

Other works include building a space- and time-efficient index used for pattern-matching in De Bruijn graphs [13] and compacting De Bruijn graphs with little memory [109]. Various other works exist [14, 49, 92, 108, 186, 224, 284, 306, 349]. Finally, for completeness, we also mention studies into probabilistic De Bruijn graphs [50, 51, 345].

Current main approaches for a compact De Bruijn graph representation are based on Bloom filters [56], a variant of Burrows-Wheeler Transform [87], and succinct data structures (§ 6).

**Schemes Based on Bloom Filters** Chikhi and Rizk [110] use a Bloom filter to store edges (with additional structures to avoid false positive edges that would affect the assembly). They traverse the graph by generating all possible outgoing edges at each vertex and testing their membership in the Bloom filter. Next, Salikhov et al. [381] design cascading Bloom filters to outperform storage requirements of Chikhi and Rizk’s approach. They change the representation of the set of false positives. The key idea is to iteratively apply a Bloom filter to: (1) represent the set of false positives, (2) then the set of “false false positives”, and so on. This cascade enables 30% to 40% less memory with respect to Chikhi and Rizk’s method [381]. Other authors used Bloom filters to implement de Bruijn graphs for pan-genomics [210] and to enhance connecting reads [418]. A redesign of the ABySS scheme was recently implemented using Bloom filters [220].

**Schemes Based on Succinct Data Structures** Conway and Bromage [125] use succinct (entropy-compressed, see § 6) data structures for a practical representation of the De Bruijn assembly graph [125]. They use [336] as succinct representations of a bitmap used to represent De Bruijn graphs. Bowe et al. [67] also incorporate succinctness. They show a representation that uses $4m + o(m)$ bits of a De Bruijn graph with $m$ edges and ensure various graph queries in constant time (for example, computing the in- and out-degree of a vertex). The structure is constructed in $O(NK \log m / \log \log m)$ time using no additional space where $K$ and $N$ and lengths of mers and the whole DNA, respectively. The authors combine (1) succinct static strings due to Ferragina et al. [169], (2) succinct dynamic strings [330], and (3) the XBW-transform structure [167]. Bowe et al.’s work was expanded by Boucher et al. [66], Belazzougui et al. [45, 46], and Pandey et al. [340]. Succinct colored De Bruijn graphs were also discussed [12, 47].

**Schemes Based on Burrows-Wheeler Transform** Various works incorporate the Burrows-Wheeler Transform for more space efficiency [35, 182, 288, 372].

4.3.2 **Grammar- and Text-Related Works.** Peshkin [350] uses the notions from both graph grammars and graph compression to understand the structure of DNA and simultaneously be able to represent it compactly. He proposes the Graphitour algorithm that finds a simplified graph to construct the input graph representing the structure of a given DNA sequence. The simplification scheme is based on contracting edges that satisfy certain criteria regarding their similarity. Next, Hayashida and Akutsu [201] use and extend Graphitour to be able to compare two different biological networks. Specifically, they assess the similarity of two networks by comparing the compression ratios of these two networks when compressed using the modified Graphitour variant. The work is applied to various metabolic networks. Finally, there are other grammar- and text-related works [186] that treat genome sequence as piece of text.
4.3.3 Hierarchical Approaches. Hierarchical approaches based on merging groups of vertices into supervertices can also be found in this domain. Brown et al. [83] consider two vertices similar if a high proportion of their neighbours are common. Such vertices are merged to form supervertices. Other similar approaches use genetic algorithms to find similar vertices efficiently [122, 433].

4.3.4 Others. Other approaches include novel types of space-efficient graphs such as Superstring graphs [94] or compressing frequent motifs in a given biological network for not only storage reductions but also faster discovery of various patterns [420].

4.4 RDF Graphs

The Resource Description Framework (RDF) is a set of World Wide Web Consortium (W3C) specifications designed to provide the semantic information in a format interpretable by machines. An RDF graph is a set of triples consisting of a subject, a predicate, and an object. Any of the triple elements can be a string; storing the triples explicitly can be memory intensive. One can thus assign identifiers to such values and use a dictionary to map them to concrete value. Consequently, to compress RDF graphs one can compress the dictionary or the underlying graph structure.

4.4.1 Modeling RDF Graphs As Relational Databases. Early approaches for compressing RDF graphs map the graphs to relational databases. One way is to simply store all RDF triples in a triple store: a table with 3 attributes (columns), an approach used in RDF storage systems such as Jena [244, 313], Sesame [82], and 3store [200]. Another approach is to use property tables. In an example scheme, several tables can be built and the attributes in each are properties shared by the triples; the remaining triples that do not fit into the property tables are stored in a triple store [424]. Third, researchers also proposed vertical partitioning [1], a scheme where there is one table per one property. The core idea is thus to group triples by predicate, generating many 2-attribute tables (one for a single predicate value). Finally, other works include space reduction schemes in the Hexastore [422], RDF-3X [333], TripleBit [432], or BitMap [32] systems. RDF-3X and BitMap use gap compression in various parts of the system; for example, RDF-3X condenses indexes in leaves of the underlying B+-tree [3].

4.4.2 Understanding and Utilizing RDF Redundancy. Pan et al. [338] first categorize the redundancy in RDF graphs into three different types: semantic, syntactic, and symbolic. Semantic redundancy can be found in RDF graphs that use more triples than necessary to describe a given set of data (i.e., they are not semantically richer than their subgraphs with fewer triples). Syntactic redundancy can be found in graphs that use excessive syntax (e.g., a plain list of triples) instead of a more compact one (e.g., binary serialization). Finally, symbolic redundancy takes place when the average number of bits needed for encoding a basic symbol (RDF resource) is not optimal. After the redundancy analysis, the authors propose to compress RDF datasets by using frequent graph patterns to remove all three aforementioned types of redundancies.

In another piece of work, Pan et al. [339] exploit the RDF graph structure to enhance compression at both the semantic and syntactic level. For the semantic level, they develop a generic framework to replace instances of the bigger graph patterns with smaller instances of the smaller graph patterns (i.e., they eliminate semantic redundancies). This approach is similar to the grammar-based compression schemes for web graphs where more complex generation rules were replaced with simpler and smaller ones. Moreover, for the syntactic level, they illustrate that the same set of RDF triples can occupy various amounts of space depending on how triples are serialised in an RDF file. They identify intra-structure redundancies (multiple occurrences of identical RDF resources within the same RDF subgraph) and inter-structure redundancies (multiple occurrences of identical resources across different RDF subgraphs).
Moreover, Fernandez et al. [161] analyzes the compressibility of RDF data sets. Specifically, the authors show that large RDF graphs can effectively be compressed because of the power law vertex degree distribution, the hierarchical organization of URLs, and the verbosity of the RDF syntax. Esposito et al. [151] develop algorithms for detecting various RDF redundancies. Pichler et al. [356] analyze the complexity of detecting redundancy in RDF datasets. Wu et al. [426] categorize RDF redundancy and design new methods for detecting redundancy. Finally, various schemes and analyses of redundancy elimination in RDF graphs were proposed and conducted by Pichler et al. [354, 355], Meier [314], Iannone et al. [215], Grimm and Wissmann [195].

4.4.3 Incorporating HDT Structure. Fernandez et al. [163, 165] design Header-Dictionary-Triples (HDT): an RDF representation that partitions RDF graph data into three modules dedicated to the RDF header information, a dictionary, and the actual triples’ structure. The modular design reduces redundancy and limits required storage by up to more than an order of magnitude. HDT takes advantage of the power law distribution in the items in RDF datasets. The size reduction is achieved due to a more condensed representation rather than a reduction in the number of triples. Next, Fernandez et al. [162] compress RDF streams by proposing the Efficient RDF Interchange (ERI) format that exploits the regularity of RDF streams. Hernandez-Illerai et al. [209] extend HDT with HDT++. HDT++ alleviates various redundancies (e.g., they group objects per predicate). They ultimately compress some popular RDF datasets by more than 50% and outperform the state-of-the-art $k^2$ trees in size by 10–13%.

4.4.4 Incorporating MapReduce. Several works attempt to reduce the size of RDF datasets with MapReduce (MR) [135]. Gimenez-Garcia et al. [185] use MR to process large RDF datasets and serialize them into the Header-Dictionary-Triples (HDT) format [165] that reduces storage overheads behind RDF graphs. Urbani et al. [417] propose to use MR to overcome the scalability problems of compressing large RDF graphs. Specifically, they use MR to construct an RDF dictionary. Similarly, Cheng et al. [104] also reduce the size of RDF graphs; they use the X10 language [98] to construct RDF dictionaries. There are other similar approaches [184, 214, 416].

4.4.5 Generating Equivalent and Smaller Rules. Joshi et al. [228–230] propose Rule Based Compression (RBC): a compression technique for RDF datasets that generates a set of new logical rules from a given dataset and removes triples that can be inferred from these rules. The authors show that RBC can prune up to 50% of the original triples without destroying data integrity. For example, a triple $<$ A, grandfather–of, C $>$ can be generated from triples $<$ A, father–of, B $>$, $<$ B, father–of, C $>$ assuming the introduction of an ontology appropriately connecting relations father–of and grandfather–of.

Fernandez et al. [166] propose a scheme called RDF Differential Stream. It uses structural similarities among items in a stream of RDF triples and combines differential encoding with zlib. Zhang et al. [436] compress RDF datasets with Adaptive Structural Summary for RDF Graph (ASSG): a compression method that uses bisimulation [146] to create an equivalent graph of smaller size where vertices with identical labels are collapsed into fewer vertices. Lyko et al. [297] use logical implications contained in the data to develop rules and simultaneously minimize the number of triples that need to be stored. Gayathri et al. [178] mine logical Horn rules [212] from the RDF datasets and then store only the triples matching the antecedent part of the mined rules. Triples matching the head part of the rules are deleted because they can be inferred by applying the rules. Guang et al. [196] propose rule-based methods to find and delete semantically redundant triples.

4.4.6 Incorporating Hierarchical Schemes. Fernandez et al. [164] compress RDF graphs by grouping triples with the common subject into adjacency lists. Then, for each RDF property value and
subject, it stores ordered IDs of the associated objects. The derived ID sequences are treated with Huffman encoding and PPMd 7-zip [342]. Next, Jiang et al. [225] propose two schemes. First, they assign a type to each RDF object and subject and then reduce the number of vertices in the RDF graph by grouping and collapsing RDF entities with the same type. Second, they compress the RDF graph by removing vertices with only one neighbor and maintaining the information on the removed vertex at its neighbor. Finally, Bazoobandi et al. [42] propose a new variant of the Trie structure [134] and use it as a dictionary for RDF datasets in a dynamic and streaming setting. They specifically alleviate common prefixes found in strings in RDF datasets.

4.4.7 Compressing RDF Dictionaries. Martinez et al. [309, 310] specifically compress RDF dictionaries. Among others, they apply existing techniques for compressing string dictionaries, including a compact form of hashing [74, 126], Front-Coding [425] (both Plain Front-Coding [74] and Hu-Tucker Front-Coding [257]), and various forms of self-indexing [74, 87, 168, 329]. Moreover, Dawelbeit and McCrindle [130] compress RDF dictionaries that are used in Google BigQuery.

4.4.8 Using $k^2$ Trees. Another recent work [18–20] combines vertical partitioning [1] with $k^2$-trees [78, 79]. The core technique is called $k^2$-triples. It first vertically partitions the dataset into disjoint subsets of pairs (subject, object), one subset per predicate. Next, these subsets of pairs are represented as binary matrices where one cell with "1" indicates that a given triple exists in a given RDF graph. These matrices turn out sparse and they are then encoded with $k^2$-trees. A related work by Alvarez et al. [22, 175] advocates Interleaved $k^2$ trees: a compressed and self-indexed representation with efficient querying of general ternary relations, similar to $k^2$ trees and their application in compressing binary relations. The main idea is to represent a given set of triples as $x$ binary relations and then use $x k^2$ trees and gather them within a single tree. Interleaved $k^2$ trees can be applied to generic ternary relations; they are evaluated on RDF. Next, Brisaboa et al. [76] propose a dynamic data structure to compactly represent binary relations; the structure is a dynamic variant of the $k^2$ tree. The above-described efforts into combining RDF datasets and $k^2$ trees were also described in theses by Alvarez [17] and Roca [132].

4.4.9 Using Succinct Data Structures. Cure et al. [128] use succinct data structures to compress RDF data and to ultimately alleviate scalability issues.

4.4.10 Others. Other efforts include the following related work. Swacha and Grabowski [402] compress RDF datasets with a combination of techniques. They separate semantic (i.e., RDF specialized) and general-purpose encoding. They also separate graph and dictionary compression, and combine various techniques over the contents (e.g., run-length encoding or reordering the content). Zneika et al. [443] summarize RDF datasets by adding the information on various instances of data patterns for more performance. Fernandez [160] compacts RDF datasets with a combination of various techniques. Jagalpure [223] designs novel indexing techniques for more scalable and storage-efficient RDF databases. Weaver and Williams incorporate a subset of the Turtle syntax [43] and Lempel-Ziv-Oberhumer (LZO) compression to reduce I/O load in parallel RDF systems. Joshi et al. [231] exploit ontology alignments and application context in RDF graphs for compression. Gallego et al. [173] focus on compressing RDF data in the context of multimedia retrieval. Deme et al. [138] enhance the design of the RDSZ scheme. Brisaboa et al. [77] propose a novel RDF storage scheme called RDFCSA that combines the data and the associated index in a single representation and builds on suffix arrays. Bit vectors for compressing RDF are used by Atre [31].
4.5 Network Graphs

Some works target compressing graphs originating in the area of networking. Gilbert et al. [183] summarize IP networks to facilitate visualization. They refer to it as the semantic graph compression to distinguish it from the algorithmic graph compression where a graph is condensed to reduce the time or space complexity of an associated graph algorithm. They preserve selected properties, for example connectivity: if an underlying graph is connected, the compressed one should also be connected. Moreover, they also develop compression schemes that collapse similar vertices into one (hierarchical schemes). The similarity measure is derived from only topological information or from vertex or edge properties included in the dataset.

Jusko et al. [232] use Bloom filters [56] to develop representations of connection graphs (e.g., P2P overlays) that reduce the amount of consumed memory; the targeted setting is Software Defined Networking (SDN) [263]. The representation enables network elements to determine which connections are to be escalated for further processing and it simultaneously prohibits extracting any other information from the graph for security. The connection graph is assumed to be dynamic.

Shi et al. [389] compress network traffic graphs by grouping motifs such as a clique into single vertices; thus again incorporating a hierarchical compression scheme.

Other works include compressing changes in network monitoring data [103].

4.6 Chemistry Networks

There are only very few works related to compressing graphs used in various chemical sciences. In general, there exist some studies on applying graph theory in chemistry [36]. Compression of such graphs was mostly not addressed. Burger et al. [85, 86] address compressing graphs used to model Super Carbon Nanotubes (SCNTs). Example such graphs are Hierarchically Symmetric Graphs (HSG) [384]. They can model hierarchical structure of SCNTs. The authors present the Compressed Symmetric Graphs (CSG) that is constructed out of the description of an HSG while exploiting the structural symmetry in the HSG to only store nodes and edges required for efficiently reconstructing requested parts of the original graph on-the-fly.

4.7 Geographical Datasets

One may distinguish two subareas in compressing graph-related datasets in geographical sciences.

4.7.1 Compressing Terrain Datasets. Geography Information System (GIS) data is usually 3D terrain data. Thus, it is stored with 3D meshes. For compressing such datasets, one could use any of the available generic schemes for mesh compression; they were covered in several surveys [300, 346, 403]. There are also works related to specifically compressing GIS data. For example, Pradhan et al. [360] target GIS data with Delaunay triangulation [180].

4.7.2 Compressing Raster Datasets. This subdomain is mildly related to graph compression, we still present it for completeness. In short, raster data is commonly used in GIS to represent attributes of the space such as temperatures or elevation measures. Now, these sets can often be represented as matrices. Ladra et al. [266, 267] use two ideas to compress such matrices: they first construct a special structure called $k^2$ raster that is based on $k^2$ trees. On top of $k^2$ trees, the nodes in the tree maintain the maximum and minimum values of each submatrix, which are associated with the representation of the raster data. This also provides the indexing functionality. Selected compact data structures are then used to encode elements of $k^2$ raster, such as the tree structure.
4.8 VLSI Graphs
Yang et al. [429] compress VLSI structures. They focus on the EDIF (Electronic Data Interchange Format) [123] data format. Their main idea is to use various data mining algorithms to discover redundancies, for example multiple identical subgraphs, and use the redundancies for compression.

5 COMPRESSION GRAPH DATABASES
We now consider works dedicated to compressing graph databases. Graph database is a database that stores graphs and enables semantic queries over them. An example of such a database is neo4j [421] or G* [265] that specifically targets compressing dynamic graphs.

5.1 Bitmap-Based Schemes
DEX [308] is a general-purpose system for managing and processing large graphs. It uses an internal representation based on compressed bitmaps [307] for efficient basic navigation operations.

5.2 $k^2$ Tree-Based Schemes
Lehmann and Perez [273] report empirical results on implementing graph queries over graph representations compressed with $k^2$ trees. They focus on two-way regular-path queries (2RPQs) as these queries can express navigating graphs with paths defined by regular expressions. Alvarez et al. [16] present a new model and representation of general graph databases, where nodes and edges are typed, labeled, and possibly attributed, and graphs may be multigraphs. They also discuss efficient implementation of graph navigation operations. Specifically, they propose the Compact Graph Database structure in which any multigraph is represented using three $k^2$ trees for three relations: (1) a relation between nodes and their attributes, (2) a relation between edges and their attributes, and (3) a relation between nodes (i.e., the actual edges). An interesting aspect of their work is a formal model of a labeled, attributed, and typed multigraph, that is a 10-tuple $(\Sigma_N, \Sigma_E, N, E, ST, \Sigma_A, NS, ES, NA, EA)$. $\Sigma_N, \Sigma_E$ are sets with node and edge types; $N, E$ are sets of pairs that associate numeric node or edge numeric identifiers with their types; $ST$ is a set of pairs that associate an edge numeric identifier with a pair of this edge source and destination nodes; $\Sigma_A$ is a set containing attribute names; $NS, ES$ are schemes that describe attributes of each node or edge type; $NA, EA$ are sets with pairs associating node or edge attributes and their values.

5.3 Succinct Data Structures
ZipG [253] is a system that transforms the input graph data into two flat unstructured files that contain the vertex and edge info, respectively. In addition, these files also store small amount of metadata together with the original input graph data for efficient interactive queries. Now, the ZipG is implemented on top of Succinct [6], a distributed compressed data store offering random access and arbitrary substring search queries on compressed unstructured data and key-value pairs.

5.4 Hierarchical Schemes
Maccioni and Abadi [298] introduce a compression scheme where certain subgraphs are collapsed to reduce the number of edges at the cost of introducing an additional vertex called a compressor vertex. The considered subgraph is identical to a 3-stage Clos topology [119] and the compression method removes the middle stage of vertices and instead introduces the additional vertex connected to all vertices in the original first and third Clos stage. This work is extended [299] to cover other subgraphs that can be replaced with sparser subgraphs.
5.5 Compressing Associated Data Structures
Several works compress various data structures related to the graph data and used to, for instance, speed up some queries or for indexing purposes. Ferragina et al. [170] compress indexing schemes for large graphs that have vertices labeled with variable-length strings. Jin et al. [226] compress transitive closures with spanning trees.

5.6 Others
Gbase [235, 236] is a graph management system that takes as input a single big file with a list of edges and partitions it into several homogeneous blocks. Second, vertices are reshuffled, i.e., they are placed in the blocks where the majority of their neighbors reside. Thanks to it, the resulting blocks are either sparse or dense. Next, Gbase compresses all non-empty blocks through standard compression such as gzip. Finally, the compressed blocks (and some meta information (e.g., the block row id) are stored in the graph databases.

6 APPROACHING STORAGE LOWER BOUNDS
The core idea behind these schemes is to encode a given graph so that the representation explicitly approaches the storage lower bound. Unless stated otherwise, all schemes operate on static graphs. Tables 2 and 3 feature the considered representations together with storage complexities.

6.1 Related Concepts
We first explain all the related notions: succinct or compact graph representations. We define them in § 6.1.1. Next, these terms are sometimes used imprecisely; we clarify such issues and provide a taxonomy of these terms used in this survey in § 6.1.2. We also illustrate the main techniques used in achieving succinctness in the majority of graph representations and others in § 6.1.3.

6.1.1 Succinctness and Compactness: Definitions. Assume \( N \) is the optimal number of bits to store some data. A representation of this data is compact if it uses \( O(N) \) bits and succinct if it uses \( N + o(N) \) bits. These definitions are used in various modern works on succinct data structures [11, 137, 323]. Many of these representations simultaneously support a reasonable set of queries fast (e.g., in \( O(1) \) or \( O(\log n) \) time) [54], others only consider reducing space complexity. Finally, some designs accelerate the process of generating (encoding) a given data representation.

Specifically, when considering an arbitrary (undirected) graph with \( n \) vertices and \( m \) edges, the number of such graphs is \( N = \binom{2n}{m} \), the storage lower bound is \( \log N \), and thus a succinct and compact representation respectively take \( \log N + o(\log N) \) and \( O(\log N) \) bits.

One can equivalently define succinctness via entropy. For example, Aleardi et al. [9, 11] state that a data structure is succinct if its asymptotic size matches the entropy of the class of represented structures and compact if it matches it up to a constant factor. In the language of the terms used above, the class of represented structures are arbitrary graphs.

6.1.2 Succinctness and Compactness: Taxonomy of Concepts. We now clarify certain issues related to succinctness and compactness. Recent works associate these terms with the definitions in § 6.1.1. Now, there exist various works that use different, although related, definitions of succinctness and compactness. For example, Galperin and Wigderson state that a succinct graph representation takes \( o(n) \) space [174]. We now describe these aspects and introduce notions that are used in the remainder of this section and in Table 2 and 3. Specifically, we use the following terms: succinct This term indicates that a representation uses the definition of succinctness from § 6.1.1 or the definition of succinctness based on entropy [9, 11] provided in the last paragraph of § 6.1.1.
Survey and Taxonomy of Lossless Graph Compression

<table>
<thead>
<tr>
<th>Reference</th>
<th>Size [bits]</th>
<th>Labels</th>
<th>Edges</th>
<th>Fast access/encoding</th>
<th>Targeted graph family (see § 2.3)</th>
<th>Scheme type (see § 6.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itai [218]</td>
<td>$\frac{1}{2}n \log n + O(n)$</td>
<td>yes</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Turan [415]</td>
<td>$\leq 12n$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Simple</td>
</tr>
<tr>
<td>Turan [415]</td>
<td>$n \log n + 12n$</td>
<td>yes</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Simple</td>
</tr>
<tr>
<td>Tamassia [406]</td>
<td>$O(n)$</td>
<td>unspl.</td>
<td>both</td>
<td>yes</td>
<td>yes</td>
<td>Planar embedding</td>
</tr>
<tr>
<td>Keeler [243]</td>
<td>$n \log n + m \log 12 + o(n)$</td>
<td>yes</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>General</td>
</tr>
<tr>
<td>Keeler [243]</td>
<td>$3n + O(1)$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Map, stick-free, loop-free</td>
</tr>
<tr>
<td>Keeler [243]</td>
<td>$m \log 12 + O(1)$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>General</td>
</tr>
<tr>
<td>Keeler [243]</td>
<td>$m \log 12 + O(1)$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Map, stick-free</td>
</tr>
<tr>
<td>Keeler [243]</td>
<td>$(3 + \log 3)m / 3 + O(1)$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Munro [324, 325]</td>
<td>$8n + 2m + o(n + m)$</td>
<td>yes</td>
<td>undir.</td>
<td>yes</td>
<td>no</td>
<td>General</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + \left(\frac{5}{2} + \frac{1}{2}\right)n + o(m + n)k$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>$t \cdot \log 3 + 3$</td>
<td>General, loop-free, $k &gt; 0$</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + \left(\frac{5}{2} + \frac{1}{2}\right)n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, loop-free</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$\frac{1}{2}m + 5n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, 3-connected</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + 3n + o(m + n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, 3-connected, loop-free</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + 2n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, 3-connected, loops</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + 2n + o(m + n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, triangulated, loop-free</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, triangulated</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$(n + m) \log 5 + 3 \log 2$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>General, 3-connected</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + \left(\frac{5}{2} + \frac{1}{2}\right)n + o(m + n)k \cdot \log 3 + 3$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, $k &gt; 0$</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + \left(\frac{5}{2} + \frac{1}{2}\right)n + o(m + n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, loops, $k &gt; 0$</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$\frac{1}{2}m + 6n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, loops</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + 4n + o(m + n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, 3-connected, loops</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + 3n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, triangulated</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + 3n + o(m + n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, triangulated, loop-free</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$2m + 2n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, triangulated, loops</td>
<td>Compact</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$(n + m) \log 3 + n + 1$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>General, 3-connected, loops</td>
</tr>
<tr>
<td>Chuang [114]</td>
<td>$(\min(n, m)) \log 3 + n + 2$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, Simple, 3-connected, loops</td>
<td>Compact</td>
</tr>
<tr>
<td>King [254]</td>
<td>$\frac{11}{7}n$</td>
<td>yes</td>
<td>unspl.</td>
<td>no</td>
<td>yes</td>
<td>Planar, triangle, loop-free</td>
</tr>
<tr>
<td>He [202]</td>
<td>$4n - 9 = \frac{1}{2}n - 1$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Plane triangulation</td>
</tr>
<tr>
<td>He [202]</td>
<td>$(\frac{1}{2} + 2 \log 3) \min{n, f} - 7$</td>
<td>no</td>
<td>undir.</td>
<td>no</td>
<td>yes</td>
<td>Plane 3-connected</td>
</tr>
<tr>
<td>He [203]</td>
<td>$\beta(n) + o(\beta(n))$</td>
<td>yes</td>
<td>both</td>
<td>no</td>
<td>yes</td>
<td>Plane triangulation</td>
</tr>
<tr>
<td>He [205]</td>
<td>$\beta(n) + o(\beta(n))$</td>
<td>yes</td>
<td>both</td>
<td>no</td>
<td>yes</td>
<td>Plane or planar</td>
</tr>
<tr>
<td>Chuang [105]</td>
<td>$2m + 3n + o(m + n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, loop-free</td>
<td>Space-efficient</td>
</tr>
<tr>
<td>Chuang [105]</td>
<td>$2m + 2n + o(n)$</td>
<td>no</td>
<td>undir.</td>
<td>yes</td>
<td>General, 3-connected, loops</td>
<td>Compact</td>
</tr>
<tr>
<td>Poulalhon [357, 358]</td>
<td>$4n$</td>
<td>unspl.</td>
<td>unspl.</td>
<td>no</td>
<td>no</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Aleardi [9, 10]</td>
<td>$2.1757 + O\left(\frac{1}{\log g} \log t\right)$</td>
<td>unspl.</td>
<td>unspl.</td>
<td>yes</td>
<td>yes</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Aleardi [9, 10]</td>
<td>$2.1757 + 36(g - 1) \log t$</td>
<td>unspl.</td>
<td>unspl.</td>
<td>yes</td>
<td>yes</td>
<td>Triangulation of a surface with genus $g$</td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$2m \log 6 + o(m)$</td>
<td>no</td>
<td>unspl.</td>
<td>yes</td>
<td>no</td>
<td>Plane triangulation</td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$t \log g + t \cdot o(\log g)$</td>
<td>no</td>
<td>unspl.</td>
<td>yes</td>
<td>no</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$t \log g + t \cdot o(\log g)$</td>
<td>no</td>
<td>unspl.</td>
<td>yes</td>
<td>no</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$n + t(\log \log g + o(\log g))$</td>
<td>edges</td>
<td>unspl.</td>
<td>yes</td>
<td>no</td>
<td>Outerplanar</td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$n + t(\log \log g + o(\log g))$</td>
<td>edges</td>
<td>unspl.</td>
<td>yes</td>
<td>no</td>
<td>General</td>
</tr>
<tr>
<td>Aleardi [11]</td>
<td>$2m + o(n)$</td>
<td>unspl.</td>
<td>unspl.</td>
<td>yes</td>
<td>yes</td>
<td>3-connected</td>
</tr>
<tr>
<td>Aleardi [11]</td>
<td>$3.24n + o(n)$</td>
<td>unspl.</td>
<td>unspl.</td>
<td>yes</td>
<td>yes</td>
<td>Triangulated</td>
</tr>
<tr>
<td>Bielloch [50]</td>
<td>$\mathcal{H}_n(n) + o(n)$</td>
<td>no</td>
<td>unspl.</td>
<td>yes</td>
<td>yes</td>
<td>Map</td>
</tr>
<tr>
<td>Yamanaka [428]</td>
<td>$6n + o(n)$</td>
<td>yes</td>
<td>unspl.</td>
<td>yes</td>
<td>no</td>
<td>Plane triangulation</td>
</tr>
</tbody>
</table>

Table 2. (§ 6.2) Compact and succinct graph representations for graphs that are planar or planar-like (maps, plane graphs, etc.). To save space, we only show the first name. "yes", "no", "edges" indicate that a graph has vertex labels, has no labels at all, has edge labels; "unspl." means labeling is not mentioned. "undir," "dir," "both" indicate that a scheme targets undirected graphs, directed graphs, or both; "unspl." means it is unspecified. *** "Fast" indicates that a given scheme attempts to reduce the time complexity of a certain query (queries) or the time to create (i.e., encode) or decode a given representation from the input graph representation (an AL or an AM); "yes" indicates that the scheme in the given row offers more efficient operations on the graph than the corresponding scheme in the below, possibly at the cost of more storage.
**Compact**

This term describes a representation that does not explicitly use the definition of succinct. An encoding of a graph is much larger than its information-theoretic tight bound, i.e., the shortest length over all possible operations on the graph than the corresponding scheme in the given row offers more efficient operations on the graph than the corresponding scheme in the below, possibly at the cost of more storage.

<table>
<thead>
<tr>
<th>Reference*</th>
<th>Size [bits]</th>
<th>Labels **</th>
<th>Edges ***</th>
<th>Fast access / encoding****</th>
<th>Targeted graph (family; see § 2.3)</th>
<th>Scheme type (see § 6.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobson</td>
<td>$O(kn)$</td>
<td>no</td>
<td>unsp.</td>
<td>yes / yes</td>
<td>$k$-page succinct</td>
<td>succinct</td>
</tr>
<tr>
<td>Cohen [120]</td>
<td>$O(n)$</td>
<td>yes</td>
<td>unsp.</td>
<td>yes / yes</td>
<td>$k$-connected “compact”</td>
<td></td>
</tr>
<tr>
<td>Munro [324, 325]</td>
<td>$2kn + 2m + o(kn + m)$</td>
<td>no</td>
<td>yes</td>
<td>$k$-page succinct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deo [140]</td>
<td>$O(n + g)$</td>
<td>yes</td>
<td>undir.</td>
<td>no / yes</td>
<td>bounded genus ($\leq g$) “compact”</td>
<td></td>
</tr>
<tr>
<td>Deo [140]</td>
<td>$O(n)$</td>
<td>yes</td>
<td>undir.</td>
<td>no / yes</td>
<td>bounded arborically separable “compact”</td>
<td></td>
</tr>
<tr>
<td>Lu [293, 294]</td>
<td>$\leq \beta(n) + o(\beta(n))$</td>
<td>yes</td>
<td>undir.</td>
<td>no / yes</td>
<td>$\text{genus } g = \left(\frac{n}{\log^2 n}\right)$ “compact”</td>
<td></td>
</tr>
<tr>
<td>Blandford [54]</td>
<td>$O(n)$</td>
<td>no</td>
<td>both</td>
<td>yes / no</td>
<td>separable compact</td>
<td></td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$n + 2m \log k + o(m \log k)$</td>
<td>no</td>
<td>unsp.</td>
<td>yes / no</td>
<td>$k$-page succinct</td>
<td></td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$n + (2 + \epsilon) \log k + o(m \log k)$</td>
<td>no</td>
<td>unsp.</td>
<td>yes / no</td>
<td>$k$-page succinct</td>
<td></td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$kn + l \log (\sigma + o(\log \sigma))$</td>
<td>edges</td>
<td>unsp.</td>
<td>yes / no</td>
<td>$k$-page succinct</td>
<td></td>
</tr>
<tr>
<td>Barbay [38, 39]</td>
<td>$n + (2m + \epsilon) \log k + o(m \log k)$</td>
<td>edges</td>
<td>unsp.</td>
<td>yes / no</td>
<td>$k$-page succinct</td>
<td></td>
</tr>
<tr>
<td>Gavoille [176]</td>
<td>$2m \log k + 4m$</td>
<td>no</td>
<td>undir.</td>
<td>no / no</td>
<td>$k$-page, $k \leq \frac{1}{2} kn / \log k$ “compact”</td>
<td></td>
</tr>
<tr>
<td>Gavoille [176]</td>
<td>$2m \log k + 4m + o(m \log k)$</td>
<td>no</td>
<td>undir.</td>
<td>yes† / no</td>
<td>$k$-page, $k \leq \frac{1}{2} kn / \log k$ “compact”</td>
<td></td>
</tr>
<tr>
<td>Gavoille [176]</td>
<td>$2m \log k + 4m + o(m)$</td>
<td>no</td>
<td>undir.</td>
<td>yes / no</td>
<td>$k$-page, $k \leq \frac{1}{2} kn / \log k$ “compact”</td>
<td></td>
</tr>
<tr>
<td>Blelloch [35]</td>
<td>$H(n) + o(n)$</td>
<td>no</td>
<td>yes</td>
<td>no / no</td>
<td>separable succinct</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. (§ 6.2) Compact and succinct graph representations for “middle-ground” and arbitrary graphs. *To save space, we only show the first name. **“yes”, “no”, or “edges” indicate that a graph has vertex labels, has no labels at all, or has edge labels; “unsp.” means labeling is not mentioned. ***“undir”, “dir”, and “both” indicate that a scheme targets undirected graphs, directed graphs, or both; “unsp.” means it is unspecified. ****“Fast” indicates that a given scheme attempts to reduce the time complexity of a certain query (or queries) or the time to create (i.e., encode) or decode a given representation from the input graph representation (an AL or an AM); “yes†” indicates that the scheme in the given row offers more efficient operations on the graph than the corresponding scheme in the below, possibly at the cost of more storage.

**succinct**

An encoding of a graph $G$ from a class of graphs $\mathcal{G}$ is described with this term if it is described as succinct and the definition does not match the one from § 6.1.1 or the one based on entropy and stated in past work [9, 11]. Example such definitions are “the length of the encoding of $G$’s representation is not too large compared to $\log |\mathcal{G}|$” [415], “the length of this encoding is not much larger than its information-theoretic tight bound, i.e., the shortest length over all possible coding schemes” [202].

**compact**

This term indicates that a representation uses the definition of compactness from § 6.1.1. This term describes a representation that does not explicitly use the definition of compactness from § 6.1.1 but directly compares to and discusses such representations [114].
We use this term when a given representation claims relationships to succinctness or compactness, and/or extensively discusses succinct or compact representations, but when it is not itself based on any precise optimality conditions. For example, Chiang et al. [105] states that a “space-efficient” representation (1) minimizes the length of a given encoding, (2) minimizes the time required to compute and decode the encoding, and (3) supports queries on this encoding efficiently.

### 6.1.3 Main Techniques for Achieving Succinctness

We now describe generic techniques for obtaining succinctness and compactness that are often used in various representations.

#### Hierarchical Decomposition

Assume that the size of a considered class of objects is $N$. For example, for a class of arbitrary graphs we have $N = \binom{n}{m}$. The high-level key idea is to divide the object to be encoded (e.g., an arbitrary graph) into small parts, group these parts in an auxiliary table, and represent them with the indices into this table. This table should contain all possible parts so that any object from a given class could be constructed from them. These parts are again divided into yet smaller (tiny) parts, stored similarly in yet other auxiliary tables. Now, the size of both small and tiny parts is selected in such a way that the sum of the sizes of all the indices and all the auxiliary tables is $O(N)$ (for compactness) and $N + o(N)$ (for succinctness) bits. The central observation that enables these bounds is that the representation consisting of small and tiny parts can be hierarchical: tiny parts only need pointers that point to other tiny parts within a single small part because small parts use other pointers that link them to other small parts.

More formally, an object to be encoded (e.g., a graph) is first split into tiny parts of size $O(\log N)$. Here, tiny means that the catalog of all these parts (i.e., an auxiliary table explicitly storing these parts) must take $o(N)$; in most cases it is $O(N \log N)$. Any tiny part is then represented with its index in this catalog and the sum of the sizes of all such indices.

Second, one must encode how these tiny parts together form the initial object. Now, the number of these parts is $O\left(\frac{N}{\log N}\right)$, the number of connections between them is $O(N)$, and a pointer to any such part takes $O(\log N)$ bits. Thus, a classical representation of the connections between tiny parts is $O(N)$, giving a compact representation.

To achieve succinctness, $\log N$ tiny parts are combined into small parts, each of which uses $O(\log^2 N)$ space. This enables using pointers of size $O(\log N)$ between small parts while tiny parts can use pointers of size $O(\log \log N)$ because they now need to point to each other only within one small part. Now, the total count of small and tiny parts is $O\left(\frac{N}{\log^2 N}\right)$ and $O\left(\frac{N}{\log N}\right)$, respectively.

This gives the total size of $O\left(\frac{N}{\log N}\right)$ bits and $O\left(\frac{N \log \log N}{\log N}\right)$ bits, respectively.

#### Parentheses Encoding

Another general succinct or compact encoding uses parentheses. To explain the idea intuitively, consider a tree and a Depth-First Search traversal of this tree. One can represent this tree with a string consisting of two parentheses, “(” and “)”. Namely, during the traversal, when a vertex is visited for the first time, an opening parenthesis “(” is appended to the string. When a vertex is visited for the second and the last time (while moving backwards in the tree structure), the other parenthesis “)” is added. Thus, a tree with $n$ vertices uses $2n$ bits (one bit per one parenthesis type). Now, a graph could be represented in a similar way. For example, if one decompose a graph into a set of spanning trees, each tree could be represented with such a string of parentheses that are in practice encoded with “0”s and “1”s.

### 6.2 Succinct and Compact Schemes

We next describe several concrete succinct and compact representations. We summarize all the considered schemes in Table 2 and Table 3. The former presents planar graphs, subclasses of planar graphs, and planar-related ones such as maps. The latter summarizes “middle-ground” graphs and
graphs of arbitrary structure; “middle-ground” are graphs that are more generic than planar ones (edges can cross outside their adjacent vertices), but do have some strong assumptions on the structure, including bounded genus, bounded arboricity, bounded number of pages, or separability. These classes are explained in § 2.3. Finally, arbitrary graphs are graphs with any structure where the only assumption can be related to the number of edges.

The majority of succinct and compact schemes listed in Tables 2 and 3 use one of a few “standard” mechanisms for achieving compactness or succinctness, described in § 6.1.3. We now group these schemes basing on the associated mechanism.

6.2.1 Schemes Based on Hierarchy. There are numerous succinct and compact representations that use the generic hierarchical way of obtaining the storage lower bounds. They include planar graphs [9, 11, 55], “middle-ground” graphs [55], and arbitrary graphs [155]. The key idea is as stated in § 6.1.3: all possible parts of any input graph (in a given class) are indexed in a lookup table and pointers of specially engineered sizes are used to ensure that any input graph can be constructed from the indexed elements to provide the desired storage bounds.

6.2.2 Schemes Based on Parentheses. Many representations use the concept of parentheses for succinct or compact encoding [176, 221, 324]. In some cases (usually planar graphs), they use one type of parentheses, but several schemes propose to use multiple types of parentheses if the input graph has a more complex structure. Moreover, several other representations use parentheses combined with reordering the vertices according to a special order called the canonical order [239]. These are all planar graphs [38, 105, 114, 202].

6.2.3 Schemes Based on Encoding Trees. Some representations are based on decomposing the input graph into trees constructed from a DFS graph traversal, and then encoding such trees using a selected scheme [243, 254].

6.2.4 Others. There are also other representations [8, 172, 218, 415, 419]. For example, Blandford et al. [54] relabel vertices based on recursive partitioning of the input graph to achieve compactness. Raman et al. [366] use succinct indexable dictionaries as a basis for ensuring succinct binary relations that can then be used to encode succinctly an arbitrary graph.

6.3 Other Storage Lower Bound Measures

Besides compactness and succinctness, there exist other concepts related to storage lower bounds that could be used while developing and analyzing compression schemes or storage-efficient representations. A detailed description of such concepts is outside the scope of this work. However, we briefly mention them to make this survey complete and to provide the associated links. One obvious related notion in discussing storage lower bounds is graph entropy; it was covered in several surveys [136, 211, 321, 394, 395]. Another way to describe storage bounds is Kolmogorov complexity of graphs that was covered in some works [204, 277, 317]. These works most often focus on investigating the “information content” of a given graph family, for example the notion of topological entropy [368, 413] is related to the probability of a graph having a certain partitioning structure. Chierichetti at el. [107] discuss the information content of web graphs and propose a graph model that reflects this content.

Some of these works specifically address compression [111–113, 198]. Choi and Szpankowski [111–113] propose the “Structural zip” algorithm for compressing unlabeled graphs; it compresses a given labeled G into a codeword that can be decoded into a graph isomorphic to G. The main idea behind the algorithm is as follows. First, a vertex v1 is selected and its neighbor count is stored explicitly. Then, the remaining n – 1 vertices are partitioned into two sets: v1’s neighbors and non-neighbors.
This continues recursively by selecting a vertex $v_2$ from $v_1$’s neighbors and storing two numbers: the number of $v_2$ neighbors among each of these two sets. Next, the remaining $n - 2$ vertices are partitioned into four further sets: the neighbors of both $v_1$ and $v_2$, the neighbors of $v_1$ that are non-neighbors of $v_2$, the non-neighbors of $v_1$ that are $v_2$’s neighbors, and the non-neighbors of both $v_1$ and $v_2$. This continues until all vertices are processed. During the algorithm execution, two types of encoded neighbor counts are maintained and concatenated into one of the separate binary sequences. First, the neighbor counts of length more than one bit (i.e., for subsets $|U| > 1$) are concatenated to form the first sequence. Second, the neighbor counts of length exactly one bit (i.e., for subsets $|U| = 1$) are concatenated to form the second sequence.

Moreover, Luczak et al. [295] design asymptotically optimal algorithms for compressing unlabeled and labeled graphs constructed with the preferential attachment model. Others analyze theoretical aspects of compressing clustered graphs [2, 27] or establish a formal relationship between the polynomials with simple zeros and the storage lower bound of a given graph [286].

6.4 Discussions on Computational Complexity

Some papers discuss the computational complexity of deriving succinct representations, for example by showing NP-hardness of some schemes [174, 341].

7 GRAPH MINIMUM ARRANGEMENT FOR STORAGE REDUCTIONS

Another line of works uses Integer Linear Programming (ILP) formulations to compress graphs by reordering vertex labels such that the new labels can be compressed more effectively. For example, some schemes assign labels to decrease differences between IDs of consecutive neighbors in each neighborhood; these minimized differences are then encoded using some variable-length coding, ultimately reducing the size of each such neighborhood and thus $G$’s size. We already discussed some schemes that aim at improving such reorderings; now we focus on existing research that explicitly uses ILP formulations or improves them. This particular problem is called Minimum Linear Gap Arrangement (MLinGapA) because it consists in minimizing linear gaps between consecutive neighbors. There are three other related problems: Minimum Logarithmic Gap Arrangement (MLogGapA), Minimum Linear Arrangement (MLinA), and Minimum Logarithmic Arrangement (MLogA). More generally, this family of problems is called Minimum Arrangement problems and are a part of a domain called Graph Layout problems [144].

7.1 Definitions of Minimum Arrangement Problems

Formally, a layout of an undirected graph $G$ is a bijective function $\phi : V \rightarrow [n] = \{1, ..., n\}$ [352] that reassigns labels of vertices so that a certain function is minimized. Now, the definition of Minimum Linear Arrangement problem (MLinA) is as follows: find a layout $\phi^*$ that minimizes the sum of differences of each pair of two vertices connected with an edge:

$$\phi^* \text{minimizes this expression} \quad \sum_{v \in V} \sum_{u \in N_v} |\phi^*(v) - \phi^*(u)| = \min_{\forall \phi, \forall v \in V} \sum_{v \in V} \sum_{u \in N_v} |\phi(v) - \phi(u)| \quad (6)$$

A strongly related problem is Minimum Logarithmic Arrangement (MLogA) where ones derives a layout $\phi^*$ that minimizes the sum of logarithms of differences; incorporating logarithms takes into account the exact bit count of numbers to be encoded

$$\phi^* \text{minimizes this expression} \quad \sum_{v \in V} \sum_{u \in N_v} \log|\phi^*(v) - \phi^*(u)| = \min_{\forall \phi, \forall v \in V} \sum_{v \in V} \sum_{u \in N_v} \log|\phi(v) - \phi(u)| \quad (7)$$
Next, the objective function can also minimize the sum of differences between consecutive neighbors in adjacency lists (Minimum Linear Gap Arrangement problem (MLinGapA)), which one can directly use to decrease the storage for a given graph if differences between vertex ID are stored and encoded with variable-length coding:

\[
\phi^* \text{ minimizes this expression} \\
\sum_{v \in V} \sum_{i=0}^{N_v-1} |\phi^*(N_{i+1,v}) - \phi^*(N_{i,v})| = \min_{\phi^* \forall v \in V} \sum_{v \in V} \sum_{i=0}^{N_v-1} |\phi(N_{i+1,v}) - \phi(N_{i,v})| 
\]

Finally, the same problem can be (analogously to MLinA) formulated including logarithms and result in Minimum Logarithmic Gap Arrangement (MLogGapA):

\[
\phi^* \text{ minimizes this expression} \\
\sum_{v \in V} \sum_{i=0}^{N_v-1} \log |\phi^*(N_{i+1,v}) - \phi^*(N_{i,v})| = \min_{\phi^* \forall v \in V} \sum_{v \in V} \sum_{i=0}^{N_v-1} \log |\phi(N_{i+1,v}) - \phi(N_{i,v})| 
\]

### 7.2 Compression Schemes Based on Minimum Arrangement Problems

There exist many works that reduce the complexity or propose heuristics for the arrangement problems in § 7.1. They are listed in existing surveys [144, 352]; we do not explicitly describe them as they do not directly relate to graph compression. Second, various works compress graphs by simply enhancing vertex labelings; we addressed many of these works in previous sections. We now only focus on works that explicitly compress graphs using minimum arrangement ILP formulations.

Safro and Temkin [379] enhance the MLogA for general graphs by incorporating link weights into the ILP formulation. They motivate it by observing that link weight can measure how often a link is used; frequently accessed links would be compressed more effectively. Their algorithm is based on a more generic strategy called the algebraic multigrid (AMG) methodology [69] for linear ordering problems [378]. In AMG, one first decomposes the original problem into several approximate ones. In the case of MLogGapA, each approximate subproblem is based on a projection of the corresponding graph Laplacian into a lower-dimensional space. Then, solutions of subproblems are used to derive the final solution. This approach has two key advantages: it has a linear complexity and can easily be parallelized and implemented with standard matrix–vector primitives. Now, Safro and Temkin first formulate the weighted MLogA problem:

\[
\phi^* \text{ minimizes this expression} \\
\sum_{v \in V} \sum_{u \in N_v} w_{vu} \log |\phi^*(v) - \phi^*(u)| = \min_{\phi^* \forall v \in V} \sum_{v \in V} \sum_{u \in N_v} w_{vu} \log |\phi(v) - \phi(u)| 
\]

Then, they conduct a series of steps, in each step they reduce the size of the input graph by **coarsening it**: repeatedly merging pairs of vertices that satisfy certain properties. At some point, the (much smaller) obtained graph is used to solve Eq. (10). Then, the original graph is derived by reversing the coarsening effects, with the computed solution updated at each de-coarsening step.

Chierichetti et al. [106] analyze various aspects of Minimum Arrangement problems; they target social networks but their formal analysis is generic. Specifically, they prove that MLogA is NP-hard on multi-graphs (graphs that admit multiple edges between two vertices). MLinGapA is NP-hard, and that MLogA has the time lower bound of $\Omega(m \log n)$ for expander-like graphs. Similarly, Dhulipala et al. [143] discuss arrangement problems in the context of graph compression; they offer a proof of the NP-hardness of MLogGapA and they introduce the ILP formulation of Bipartite Minimum Logarithmic Arrangement (BiMLogA), essentially the MLogA for bipartite graphs.
Finally, there exist various algorithms that enhance graph compression by relabeling vertices but without explicitly mentioning the ILP formulation of arrangement problems [54, 57, 60, 61]. We covered them extensively in past sections.

8 REMAINING SCHEMES

We also discuss schemes that fall outside other categories. Johnson et al. [227] discusses how to compress binary matrices by reordering the columns so that the whole matrix is more compression-friendly. Such matrices could be used to represent graphs using less storage. Moreover, Borici and Thomo [65] compress graphs by transforming them into corresponding hypergraphs and then partitioning the hypergraphs so that vertices with similar properties (e.g., degrees) are in the same partition. This makes the corresponding adjacency matrix more compression-friendly. Other works include compressing dense graphs [240] and vertex-transitive graphs [287].

8.1 Hierarchical Schemes

We discuss general hierarchical schemes similar to those presented in the web graph section § 4.1.5.

8.1.1 Grouping Cells of Adjacency Matrix. First, we outline works that utilize hierarchy related to adjacency matrices, for example, group non-zero cells into blocks and compress such blocks separately. Lim, Kang, and Faloutsos [234, 282] propose SlashBurn: a scheme that exploits high-degree vertices (hubs, found often in real-world graphs) and their neighbors (spokes) to achieve high compression ratios. This forms a different type of community structure than the traditional “caveman” communities where vertices are clustered within certain groups (“caves”) and sparsely connected to other vertex groups. They propose vertex relabeling that uses this observation and results in space-efficient representation of the adjacency matrix. The SlashBurn algorithm (1) removes high-degree vertices and assigns them the lowest labels (2) finds connected components in the resulting graph and assigns the vertices in these components the highest labels, in the decreasing order of the sizes of the connected components that they belong to, (3) finds the giant connected component in the resulting graph and executes step (1) on it recursively, until its size is below a certain threshold. SlashBurn was extended to distributed-memory settings. Moreover, Li and Rao compress graphs by grouping parts of the adjacency matrix and using different codes to reduce the space required to store a given group [275]. Furthermore, Li et al. [276] first cluster graph adjacency matrix via graph structure information, and then represent the clustered matrix by lists of encoded numbers. Finally, various schemes described in other parts of this survey are related to compressing adjacency matrices hierarchically. Examples are works on $k^2$-trees [78] (see § 4.1.3).

8.1.2 Schemes Based on Supervertices. A large portion of hierarchical schemes explicitly groups vertices with similar properties into supervertices (also called supernodes) and collapse edges between them into superedges. Many of them were described in § 4.1.5. Here, we mention works that are not explicitly related to web graphs. Stanley et al. [399] find clusters in a given graph and then simplify and represent it using supervertices with one vertex being formed from one cluster. Next, Toivonen, Zhou, and others [412, 441] propose a hierarchical scheme that targets weighted graphs. They group vertices with similar neighborhoods into supervertices, and group multiple weighted edges between such supervertices into superedges. Another similar work that considers algorithms for all-pairs shortest paths, bipartite matching, and edge and vertex connectivity was conducted by Feder and Motwani [157, 158]. Moreover, Brown et al. [83] use genetic algorithms to assess the similarity of vertices (where two vertices are considered similar if many of their neighbors are identical). Similar vertices are merged into supervertices and the graph size is ultimately reduced. In addition, Sun et al. [401] measure the overlap of neighbors between vertices and, if the overlap
is large enough, the identical neighborhood parts are collapsed and a certain data structure is used
to encode this structural change. Lamarche-Perrin et al. [269] target compressing weighted graphs
with supervertices. Finally, Nourbakhsh simplifies the input graph (and thus reduces its size). He
uses Szemeredi’s Regularity Lemma [259] to cluster the graph and to produce a smaller graph
where clusters become vertices [334].

8.1.3 Tree Decompositions. Some lossless compression schemes decompose a graph into several
trees, encode these trees separately, and ultimately reduce the overall space requirements. Chen
and Reif [102] decompose an input graph into several binary trees, and finally compress these
trees with a proposed tree-compression algorithm. The key idea in compressing a single binary
tree is to further decompose this tree into smaller subtrees. These subtrees are small enough that
any such subtree can be found multiple times in the input tree. Thus, after the full binary tree
decomposition, the authors calculate occurrence probabilities for each subtree and assign the
corresponding Huffman code to it. Finally, the tree is encoded by traversing it and assigning the
above codes. Now, the method to find and count respective subtrees is similar to counting words
in texts. Specifically, the authors traverse the input tree with BFS and build a suffix tree in the
process where each node of a suffix tree corresponds to one specific subtree. A similar approach
for compressing probabilistic graphs was described by Maniu et al. [305].

8.1.4 Others. Approximation algorithms for finding the best virtual-node hierarchical compres-
sion were proposed by Feder et al. [156]. They also illustrated that the optimal compression of this
type is NP-hard. Other works include compression used for obtaining better clustering [115, 322],
using quadtrees to compress adjacency matrices [99], compressing graphs that model automata [319],
partitioning an input graph and compressing each partition independently [142].

8.2 Compression for More Efficient Computation

Here, we outline works that specifically use compression for faster graph algorithms. These schemes
are different from the ones described in the section devoted to problem-aware graph compression
(§ 10.1) because they do not propose novel compression but discuss how to use existing compression
schemes for faster graph algorithms. Liakos et al. [279] use various compression techniques (bit
vectors and different types of coding techniques) in distributed-memory graph processing engines
to reduce the pressure on the memory subsystem and thus accelerate processing. Next, Granskog
and Striger analyze whether graph traversal algorithms (BFS, DFS) can be accelerated by using
compression methods such as \( k^2 \)-trees [193]. Other works use compression as one of the tools
for better data mining capabilities [159], faster queries on graphs [326], accelerating subgraph
matching by reducing the size of sets that contain matching candidates [361], or solving bin packing
problems more efficiently [68].

8.2.1 Compression in Graph Processing Engines. Some works specifically discuss how to accel-
erate a given graph processing engine with compression. Shun et al. [392, 392] developed Ligra+,
an enhancement over the Ligra graph processing engine [391] that uses parallelism to accelerate
compression and decompression of graph data and thus amortize the costs of utilizing compressed
graph representations while reducing the pressure on the memory subsystem. Other works that
use parallelization to accelerate compression also exist [149]. Furthermore, Chen et al. [101] used
compressed graphs with a generic topological OLAP framework in online graph analysis. Another
paper [4] uses various vertex relabelings for more compression friendly graph layout within Em-
ptyHeaded, an engine that outperforms standard OLAP systems. Chavan conducted an empirical
study on graph compression in engines such as Pregel [100, 301].
8.3 Vertex Coding
Certain papers from 60s are tentatively connected to graph compression. Specifically, Breuer and Folkman [70, 71] analyzed coding vertices, i.e., assigning each vertex a unique binary code that is always smaller than a certain constant if two vertices are connected, and is always larger than this constant if two vertices are not connected. Using these coding schemes, the adjacency of any two nodes can be determined by the Hamming distance of their labels and may have applications in the domain of implicit graph representations (§ 9.2).

9 RELATED DOMAINS COVERED IN SURVEYS
We now mention works and surveys covering areas that are related to lossless graph compression. First, there are works on compressing graphs with the purpose of more effective visualization, for example Dwyer et al.’s [147]. They were partially covered in another survey [291]. Second, compression of meshes was covered extensively in several surveys [300, 346, 403]. Third, compression of trees is outside the scope of this work. It is partially covered in other works [242].

9.1 Lossless Summarization of Graphs
Summarization of graphs is an area where input graph data is summarized in order to provide a smaller graph description that may focus on some particular graph aspects [30, 249–252, 292, 371, 386]. These works were covered in a survey [291]. The most important connection to graph compression is that in many of these schemes, the process of graph summarization also leads to size reduction. For example, vertices within a cluster are grouped to form a supervertex, and edges are merged into superedges [44, 260, 261, 272, 289, 290, 343, 370, 410, 427, 431, 439, 442], similarly to many hierarchical schemes in web graphs (§ 4.1.5). Some works use or discuss bisimulation, especially in the domain of RDF graphs [89, 95]. In addition, various works use summarization to better understand the structure of graphs in domains such as biology [331, 411] or independently of a specific domain [7]. Moreover, there are works dedicated to the summarization of dynamic graphs [248, 362, 414].

9.2 Efficient and Implicit Graph Representations
Intuitively, efficient (in many cases also called implicit) graph representations provide vertex labels that encode the structure of the input graph so that no additional structure dedicated to storing edges is required. For example, Kannan and Naor in their seminal work [238] assign $O(\log n)$ bit labels to vertices such that these labels completely encode the structure of the graph. Thus, no additional data structure that determines edges is required. In addition, given the labels of any two vertices, the authors show that one can test if the vertices are adjacent in time linear in the size of the labels. Many other such schemes exist [15, 344, 382, 404]. Another thread of related work are algorithms for efficient derivation of such representations [26]. Now, such representations are covered extensively in a book by Spinrad [398]. Since the book was published, more such representations were discovered [97, 129, 177, 405].

Terminology Clarification We now clarify a certain terminology issue. An implicit graph representation as described above is a representation where vertex labels themselves encode the information on edges between vertices. Now, the term implicit is used in another context in the literature [137]. It describes a representation of an arbitrary data that is a constant additive factor away from the storage lower bound for this data. Formally, if the optimum to store some data is $N$ bits, an implicit representation takes $N + O(1)$ bits [137]. We do not know of any graph representations that are implicit in the second sense.
10 TAXONOMY AND DISCUSSION OF FEATURES

We now group and discuss graph compression schemes based on selected common features for better understanding of lossless graph compression.

10.1 Problem-Aware Graph Compression

First, we discuss schemes that, despite applying compression, still allow to obtain selected graph properties or solve selected graph problems fast. Sadri et al. [377] propose Shrink, a compression scheme that preserves distances between vertices. The compression proceeds in steps, in each step it iteratively merges vertices. During each merging, a system of linear equations is solved to define new edge weights to minimize changes in the distances. Merging continues until a specified number of vertices is reached. Moreover, Fan et al. [152, 153] develop compression strategies that preserve high performance and losslessness for two classes of graph queries: reachability and graph pattern queries via (bounded) simulation. Next, Hernandez discusses application-driven graph representations and compression [205]. Another similar work (performed for general graphs) that considers algorithms for all-pairs shortest paths, bipartite matching, and edge and vertex connectivity was conducted by Feder and Motwani [157, 158]. Finally, most of succinct and compact graph representations provide graph queries that ensure a specific time complexity, most often constant-time or logarithmic, see § 6.

10.2 Compression of Dynamic Graphs

We separately discuss compressing dynamic graphs (in the literature, they are also called temporal, evolving, or time-evolving). Brodal and Fagerberg [80] present a linear space graph data structure for graphs with bounded arboricity (example such graphs are planar graphs or bounded-treewidth graphs) under insertions, edge deletions, and adjacency queries. The proposed representation is the adjacency list representation, augmented with a simple scheme that maintains the structure under graph modifications. The core idea in proving the stated time bounds (constant-time adjacency queries in a graph with bounded arboricity $c$) is to reduce this problem to a simpler problem of assigning directions to edges (i.e., constructing a directed graph out of the input undirected one) so that all vertices have outdegree $O(c)$.

Iverson and Karypis [219] propose five data structures for representing dynamic sparse graphs. Their structures offer different trade-offs between size and speed of provided graph operations. The structures are: Linked-List (LL), Batch Compressed Sparse Row (BCSR), Dynamic Adjacency Array (DAA), Dynamic Intervalized Adjacency Array (DIAA), and Dynamic Compressed Adjacency Array (DCAA). LL is based on a simple set of linked lists with one list being responsible for one vertex neighborhood. BCSR is essentially an LL, but when a size of a linked list grows too large, it is resized into a static CSR. In DAA, each neighborhood is a dynamically allocated array that must be resized if there are updates. DIAA is essentially a DAA enhanced with storing contiguous vertices as intervals. Finally, DCAA leverages ideas from the WebGraph framework (§ 4.1.4).

Other works on dynamic graphs exist, for example Boldi et al. [59] analyzes how the web graph evolves and how its respective snapshots can be compressed, Caro et al. [90] design compact graph representations that enable answering queries fast, and Klitzke and Nicholson [256] mention compressing dynamic graphs as a part of their general framework for managing dynamic succinct data structures.

Finally, several dynamic schemes were described in the other sections of this survey, for example in the parts devoted to summarizing dynamic graphs [248, 362, 386, 414] (§ 9.1), compressing RDF graphs (§ 4.4), De Bruijn graphs [45] (§ 4.3.1), graph databases [265] (§ 5), succinct data
structures [406] (§ 6), and others [76, 320, 383]. We list them to facilitate navigating the survey and refer the reader to these specific sections for more information.

Dynamic graphs are also considered in streaming settings; we discuss this separately in § 10.3.

10.2.1 Viewing Graphs As Tensors. We separately discuss works that add more dimensions to its adjacency matrix to model changes. Caro et al. [91] represent dynamic (temporal) graphs using 4-dimensional binary tensors. Two dimensions are used to model edges and two other dimensions model time intervals where a given edge exists. Then, they propose to compress such a representation with a generalization of $k^2$-trees [78] (see § 4.1.3) to a $d$-dimensional space, called $k^d$-tree. The key idea is similar to that of simple $k^2$-trees, namely, parts of a $d$-dimensional tensor with zeros in cells are compressed with internal nodes of a $k^d$-tree while tree leaves represent parts of the tensor that have more than one non-zero cell. Related approaches based on viewing a dynamic graph as a 4-dimensional object were discussed by Brisaboa, Bernardo, Caro, and others [75, 131].

10.3 Compression of Graphs in Streaming Settings
Various compression schemes are designed for streaming settings [139, 246, 247, 285, 332, 337, 385, 407, 408, 437]. For example, Nelson et al. [332] use quadtrees to compress graph streams.

11 CONCLUSION
Graph compression is an important area of research as it can be used to accelerate numerous modern graph workloads by reducing the amount of transferred data. Yet, it is a diverse set of fields driven by different communities, with a plethora of techniques, algorithms, domains, and approaches. We present the first survey that analyzes the rich world of lossless graph compression. We do not only list and categorize the existing work, but also provide key ideas, insights, and discuss formal underpinning of selected works. Our work can be used by architects and developers willing to select the best compression scheme in a given setting, graph theoreticians aiming to understand the high-level view of lossless graph compression, and anyone who wants to deepen their knowledge of this fascinating field.

REFERENCES


