Faster Graph Algorithms Through DAG Compression

Max Bannach
European Space Agency, Advanced Concepts Team, Noordwijk, The Netherlands

Florian Andreas Marwitz
Institute of Information Systems, Universität zu Lübeck, Germany

Till Tantau
Institute for Theoretical Computer Science, Universität zu Lübeck, Germany

Abstract

The runtime of graph algorithms such as depth-first search or Dijkstra's algorithm is dominated by the fact that all edges of the graph need to be processed at least once, leading to prohibitive runtimes for large, dense graphs. We introduce a simple data structure for storing graphs (and more general structures) in a compressed manner using directed acyclic graphs (DAGs). We then show that numerous standard graph problems can be solved in time linear in the size of the DAG compression of a graph, rather than in the number of edges of the graph. Crucially, many dense graphs, including but not limited to graphs of bounded twinwidth, have a DAG compression of size linear in the number of *vertices* rather than *edges*. This insight allows us to improve the previous best results for the runtime of standard algorithms from quasi-linear to linear for the large class of graphs of bounded twinwidth, which includes all cographs, graphs of bounded treewidth, or graphs of bounded cliquewidth.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Data structures design and analysis; Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases graph compression, graph traversal, twinwidth, parameterized algorithms

Digital Object Identifier 10.4230/LIPIcs.STACS.2024.44

Funding *Florian Andreas Marwitz*: The research for this paper was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2176 "Understanding Written Artefacts: Material, Interaction and Transmission in Manuscript Cultures", project no. 390893796. The research was conducted within the scope of the Centre for the Study of Manuscript Cultures (CSMC) at Universität Hamburg.

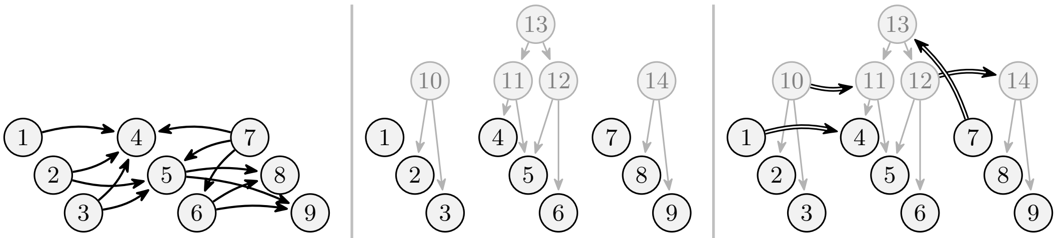
1 Introduction

Graph traversal or graph searching is a fundamental subroutine in algorithmic graph theory. Given a directed graph (digraph) and a source vertex, the task is to explore the graph following a predefined strategy. Two famous incarnations of such algorithms are *depth-first search* (DFS) and *breadth-first search* (BFS), which, as the names suggest, explore the graph by following long paths first or by unraveling the graph layer by layer. Both algorithms have a broad range of applications, including the computation of connected components or a topological ordering of the input, identifying separators, testing whether the input is planar, finding shortest paths, computing maximum flows, and many more [10]. They are also central in more applied fields and, for instance, are a crucial building block in garbage collection [8], artificial intelligence [15, 20], and web crawling [7, 9]. It is well-known that both algorithms can be implemented in time $O(m)$, where m is the number of edges of the input graph [12, Chapter 5.5]. In particular, for the class of *sparse* graphs, where $m = O(n)$, both algorithms run in time linear in the number of vertices. Many (but by far not all) natural graph classes are sparse, including planar graphs, d-degenerate graphs, series-parallel graphs, or graphs of bounded treewidth [17]. In contrast, for very *dense* graphs, where $m = \Omega(n^2)$, until recently, only relatively trivial examples (such as cliques) were known for which these problems could be solved in time $o(n^2)$.

In this paper, we propose a simple data structure, dubbed *DAG compression,* that will prove useful in computing a BFS or DFS on dense graphs $G = (V, E)$. The idea is to represent complete bipartite subgraphs of G by storing *compressed edges,* which are just pairs of vertices of a DAG. Formally, a *cluster DAG for G* is a directed acyclic graph $C = (V', A)$ whose *sinks are exactly the vertices in V* and each vertex $v' \in V'$ represents a *cluster $C(v')$,* which is *the set of sinks reachable from v' in C.* A *compressed edge* is a pair $(u', v') \in V' \times V'$ that encodes that there are edges in G from each vertex in $C(u')$ to each vertex in $C(v')$; and to encode all edges in G, we use a *compressed (edge) relation $E' \subseteq V' \times V'$* such that $E = C(E') := \bigcup_{(u',v') \in E'} C(u') \times C(v')$, see Figure 1 for an example. In case G contains multiple edge relations $E_1, E_2, \ldots, E_k$ (like red edges and blue edges), we compress each E_i separately using a compressed relation E'_i (but using the same cluster DAG).

Crucially, we will show that BFS and DFS can be implemented in a way such that their time complexity is linear in the *total number $|A| + |E'|$ of edges in the DAG compression* (called the *size* of the DAG compression in the following) and *no longer necessarily linear in the number $|E|$ of edges of the original graph.* Thus, whenever we can find a DAG compression of a graph whose size is linearly bounded by the number n of vertices in the original graph, we can lower the runtime of BFS and DFS from $O(m)$ to $O(n)$.

A powerful motivation for studying DAG compressions comes from its relation to the prominent *class of graphs of bounded twinwidth.* Twinwidth is a structural graph parameter introduced in 2020 by Bonnet et al. [5] to measure the distance of a graph from being a cograph (detailed definitions will be given later). The importance of this parameter lies in the fact that for many graph classes commonly studied in the literature this parameter is bounded (so the class of graphs of bounded twinwidth is large), but the model checking problem for first-order logic on structures of bounded twinwidth is still fixed-parameter tractable (so many problems are still in FPT on this class). However, graphs of bounded twinwidth are not only interesting in the context of powerful logical characterizations and algorithmic meta-theorems: It was recently shown [4] that in unweighted graphs of bounded twinwidth, the *single-source shortest path* problem (SSSP) can be solved in time $O(n \log n)$, despite the fact that such graphs can easily have $\Omega(n^2)$ edges. Consequently, the *diameter* of



Graph $G = (V, E)$, black edges are in E

Cluster DAG $C = (V', A)$, gray edges are in A

DAG compression (V', A, E'), compressed edges in E' depicted as double lines

Figure 1 Example of how a DAG compression works: For the graph G with vertex set $V = \{1, \dots, 9\}$, one possible cluster DAG is the shown C with vertex set $V' = \{1, \dots, 14\}$. Note that the sinks of C are exactly the vertices in V. The cluster edges of C, shown as straight gray lines, determine subsets of V ("clusters") via reachability: The cluster $C(10)$ of vertex 10 is the set $\{2, 3\}$; and $C(11) = \{4, 5\}$, $C(12) = \{5, 6\}$, $C(13) = \{4, 5, 6\}$, $C(14) = \{8, 9\}$ and note that, for instance, $C(1) = \{1\}$. Pairing C with a relation $E' \subseteq V' \times V'$, shown as double lines, yields a DAG compression of G: Each edge $(u, v) \in E'$ adds $C(u) \times C(v)$ to E. For instance, the edge $(12, 14) \in E'$ implies that in E there is a complete bipartite graph with shores $C(12) = \{5, 6\}$ and $C(14) = \{8, 9\}$.

a graph of bounded twinwidth, i.e., the maximum length of a shortest path, can be computed in time $O(n^2 \log n)$, while it is also known [4] that it can*not* be computed in time $O(n^{2-\epsilon})$ unless the strong exponential time hypothesis fails. Hence, there is a $\log(n)$-gap between the known lower and upper bounds for determining the diameter of graphs of bounded twinwidth.

One of our main results will be that *graphs of bounded twinwidth admit a linear-size DAG compression.* Combining this with our BFS implementation that runs in time linear in the size of the DAG compression, we see that on graphs of bounded twinwidth, one can actually solve SSSP in time $O(n)$ and, thus, can solve the diameter problem in time $O(n^2)$. In particular, we close the gaps in the runtime left open by previous work.

However, one has to be careful regarding the exact claims of the just-mentioned results: All known algorithms working on graphs of twinwidth d, including our algorithm for computing a linear-size DAG compression of a graph of bounded twinwidth, need access to a so-called *contraction sequence.* Such a sequence is a linear-size witness that a graph has twinwidth d and, indeed, they can be used to show that deciding whether a graph has twinwidth d lies in NP (and it is known [3] that at least for $d = 4$ the problem is NP-complete). For instance, the algorithm from [4] for the diameter problem needs access to a contraction sequence witnessing a twinwidth of d, in order to run in time $O(d \cdot n^2 \log n)$; but the lower bound of $O(n^{2-\epsilon})$ also still holds when a contraction sequence is given.

Our Contributions. Our first main contribution is conceptual: We propose the already mentioned *DAG compression* data structure and study their basic properties. The *size* of these compressions (the number $|A|$ of cluster edges plus the number $|E'|$ of compressed edges) will be of particular interest since we will show that important problems can be solved in a time that is linear with respect to this size.

The central tool underlying our algorithms is a construction that uses a DAG compression $D = (V', A, E')$ of a graph $G = (V, E)$ to build an edge-weighted graph $S = (V'', E'', w'')$ with $V \subseteq V''$ and $w'' \colon E'' \to \{0, 1\}$, called the *switching graph of D* (since paths in this graph repeatedly switch between two parts of it, see Figure 2 for an example).

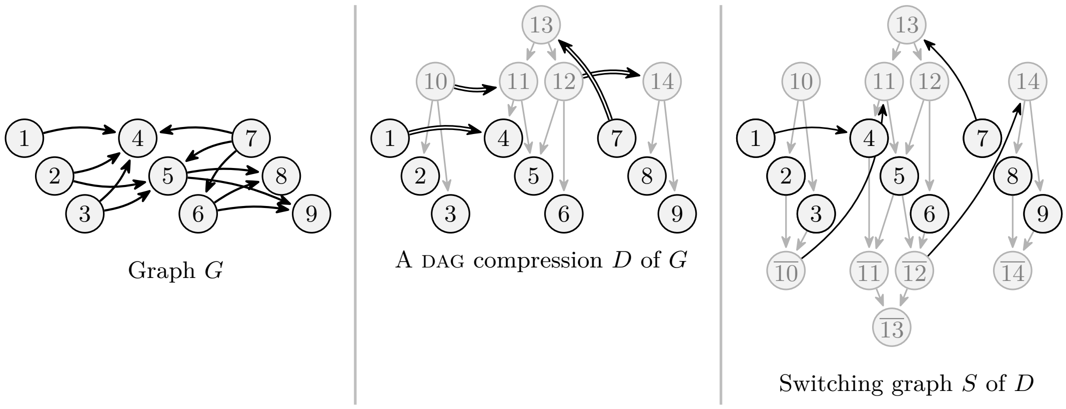


Figure 2 The example graph $G = (V, E)$, DAG compression $D = (V', A, E')$, and cluster DAG $C = (V', A)$ from Figure 1. The switching graph $S = (V'', E'', w'')$ results from first taking the disjoint union of C and the copy $\bar{C}$, where all edges are reversed, and unifying the vertices in V. This results in $2|A|$ many edges (shown in gray in S above) and we set their weight to 0. In addition, for each edge $(u', v') \in E'$ there is a *switching edge* $(\bar{u}', v')$ in E'', shown in black, that leads from the lower part to the upper part and has weight 1. A path in G of length 2, like the path $3 \to 5 \to 9$, corresponds to a path $3 \to \overline{10} \to 11 \to 5 \to \overline{12} \to 14 \to 9$ in S of weight 2 as it contains two switching edges (black edges of weight 1).

▶ **Theorem 1.1.** *Let $G = (V, E)$ be a directed graph, let $D = (V', A, E')$ be a DAG compression of G, and let $S = (V'', E'', w'')$ be the switching graph of D. Then for every pair $(u, v) \in V \times V$ we have $d_G(u, v) = d_S(u, v)$, that is, the distance from u to v is the same in G and in S.*

Since it will be immediate from the construction that the number $|E''|$ of edges in the switching graph is $2|A| + |E'|$ and thus at most double the size of the DAG compression D, we easily get fast DFS and BFS algorithms for graphs that admit a DAG compression of linear size:

▶ **Theorem 1.2.** *On input of a DAG compression $D = (V', A, E')$ of a graph $G = (V, E)$ we can visit the vertices in V both in BFS and DFS order in time $O(|V'| + |A| + |E'|)$.*

As graphs with bounded twinwidth have a linearly bounded dag compression, a direct consequence of Theorem 1.2 is that we close the gap between the lower and upper bound for computing the diameter of graphs of bounded twinwidth:

▶ **Corollary 1.3.** *On input of a contraction sequences that witnesses that a graph G has twinwidth at most d, we can compute the diameter of G in time $O(d \cdot n^2)$.*

Access to a fast depth-first search allows us to implement other operations from algorithmic graph theory in time $O(n)$. For instance, the strongly connected components of a digraph can be computed by two consecutive depth-first searches using Kosaraju's algorithm [22]:

▶ **Corollary 1.4.** *On input of a DAG compression $D = (V', A, E')$ of a graph $G = (V, E)$, the strongly connected components of G can be computed in time $O(|V'| + |A| + |E'|)$.*

Another traditional application of the depth-first search is the detection of cycles in directed graphs as well as the computation of a topological sort of the input [23]:

▶ **Corollary 1.5.** *On input of a DAG compression $D = (V', A, E')$ of a graph $G = (V, E)$, we can test in time $O(|V'| + |A| + |E'|)$ whether G contains a cycle and, if not, compute a topological sorting of G.*

It is well-known that a BFS can compute the shortest path between two vertices in *unweighted* graphs. However, this is different in *weighted* graphs, in which a more refined algorithm must be used. Generalizing the DAG compression to weighted graphs, we obtain:

▶ **Theorem 1.6.** *On input of a DAG compression $D = (V', A, E', w')$ of a weighted graph $G = (V, E, w)$ with $w\colon E \to \mathbb{N}$, the single-source shortest path (SSSP) problem can be solved in time $O\big((|V'| + |A| + |E'|)\log(|V'| + |A| + |E'|)\big)$.*

Related Work. Many graph compression methods are known in the literature; the one most similar to ours is by Toivonen et al. [26]. They also introduce supernodes and superedges with the idea that an edge between two supernodes represents all edges between vertices within these supernodes. However, they partition the vertex set into a set of supernodes, whereas our compression allows for nested vertex combinations. The representation of Navlakha et al. is similar to the one of Toivonen et al. with an additional set of edge corrections, i.e., edges that must be deleted or added to retrieve the original graph [18]. Tian et al. provide two operations: One for creating a graph compression based on user-given attributes and another to further control the compression [25]. Zhang et al. further refine this compression to include numerical attributes and add more automation [27].

Using distance-equivalent graphs to speed up routing algorithms is commonly done in theory and practice [24]. However, the objective is usually to replace a large input graph with a smaller graph in which distances are approximately the same as in the input. In contrast, our use of distance-equivalent graphs does not involve any approximations: The distances in the switching graphs are precisely as in the original graph.

Twinwidth was introduced in 2020 for graphs and digraphs by Bonnet et al. [5] and interest quickly increased as witnessed by dozens of new research papers each year since then. One of the earliest and most remarkable results is an fpt-algorithm for the model-checking problem of first-order logic [5]. Graphs of twinwidth 0 and 1 can be recognized in polynomial time [28, 14], but deciding whether a graph has twinwidth at most 4 is NP-complete [3]. Besides the aforementioned meta-theorem, dedicated dynamic programs are known to compute maximum cliques, independent sets, and minimal dominating sets on graphs of bounded twinwidth [4]. It is also known that all triangles of a graph of twinwidth at most d can be counted in time $O(d^2n + m)$ if a corresponding contraction sequence is given [16]. Ahn et al. [2] study the twinwidth of random graphs. Schidler and Szeider provide the first practical strategies to compute contraction sequences using a SAT-solver [21] and Ganian et al. show that weighted model counting can be done efficiently on formulas of small twinwidth [13]. Bonnet et al. introduced twin-models [6], which can also compress graphs and is similar to our result in Theorem 4.4. However, one main thrust of defining DAG compressions is the usefulness independently of twinwidth, which is also one of the reasons we consider DAG compressions rather than tree compressions.

Structure of this Paper. We define DAG compressions and have a look at some basic properties and operations in the next section. In Section 3 on algorithms, we define the switching graph and show how it can be used to implement fast versions of BFS and DFS, and related algorithms. In Section 4, we show how we can build linear-size DAG compressions. Our particular focus will be on graphs of bounded twinwidth, where we turn a given contraction sequence into a DAG compression of linear size.

2 DAG Compressions: Definition, Examples, and Basic Constructions

The idea behind DAG compressions is – as already pointed out in the introduction – to compress complete bipartite subgraphs of a given graph by single "compressed edges" that link vertices of the cluster DAG. The cluster DAG has the job of encoding sets of vertices via the reachability relation: Each vertex of the cluster graph encodes all sinks that are reachable from it. In the following, we formalize these ideas and give examples. We also show how basic update and construction operations on DAG compressions can be implemented.

Basic Terminology. Before we proceed, let us fix some terminology and notation: To simplify the presentation, a *graph* is always a pair (V, E) consisting of a non-empty finite set V of vertices together with a relation $E \subseteq V \times V$. In other words, by "graph" we always refer to a simple, non-empty, directed graph; undirected graphs are just directed graphs with a symmetric edge relation. Throughout this paper, n will refer to the size $|V|$ of the graph G currently under consideration and m will refer to $|E|$.

An *(edge-)weighted graph* is a triple (V, E, w), where $w\colon E \to \mathbb{N}$ maps edges to nonnegative integers. The weights are *binary* if $w(e) \in \{0, 1\}$ holds for all $e \in E$. An unweighted graph can also be seen as a weighted graph in which all weights are 1. A *walk of length l* in a graph G is a sequence $(v_0, \dots, v_l)$ of vertices such that $(v_{i-1}, v_i) \in E$ holds for all $i \in \{1, \dots, l\}$. For $s = v_0$ and $t = v_l$, the walk is also called an *s-t-walk* and we say that *t is reachable from s*. The *weight* of a walk is the sum $\sum_{i=1}^{l} w\big(v_{i-1}, v_i\big)$. Note that for unweighted graphs the length and the weight of a walk are the same.

A walk is called a *path* if all vertices are distinct. A walk is called a *cycle* if $l \geq 3$, $v_0 = v_l$, and $(v_0, \dots, v_{l-1})$ is a path. The *distance function for G* is the function $d_G\colon V \times V \to \mathbb{N} \cup \{\infty\}$ that maps each pair (u, v) of vertices to the minimum weight of any walk u-v-walk in G (or to ∞, if no such walk exists).

A graph is a *directed acyclic graph* (a DAG) if there is no walk in G of length at least 1 with $v_0 = v_l$. A *sink* in a DAG is a vertex $s \in V$ of out-degree 0, that is, without edges leaving s. Note that a DAG must always have at least one sink.

2.1 Definition of DAG Compressions and Examples

In order to formalize the notion of DAG compressions, we start with cluster DAGs:

▶ **Definition 2.1** (Cluster DAGs and Compressed Edges). *A* cluster DAG *for a set V is a DAG $C = (V', A)$ such that V is exactly the set of sinks of C. Given a vertex $v' \in V'$, the* cluster *$C(v')$ of v' is the subset of V of all sinks that are reachable from v' in C. A pair $(u, v) \in V' \times V'$, not necessarily an element of A, is called a* compressed edge.

▶ **Definition 2.2** (DAG Compression). *Let $G = (V, E)$ be a graph. Let $C = (V', A)$ be a cluster DAG for V. A* DAG compression *of G is a triple $D = (V', A, E')$, where $E' \subseteq V' \times V'$ is a* compressed (edge) relation, *such that $E = \bigcup_{(u',v') \in E'} C(u') \times C(v')$. The* size *of D is the number $|A| + |E'|$.*

We already gave an example of a DAG compression of a graph in Figure 1. In the following we consider three more examples in order to explain the concept.

▶ **Example 2.3** (No Compression). A trivial way of compressing any graph $G = (V, E)$ is to do no compression at all, that is, to use (V', A, E') with $V' = V$, $A = \emptyset$, and $E' = E$. Note that, indeed, if there are no edges in the cluster DAG, each vertex is a sink.

This trivial example shows that we can always come up with a DAG compression of size m for any graph G. In particular, for any class $\mathcal{C}$ of graphs that has only a linear number of edges (that is, for which there is a constant c such that for all $(V,E) \in \mathcal{C}$ we have $|E| \le c \cdot |V|$), all graphs in $\mathcal{C}$ admit linear-size DAG compressions. A prominent example of such classes are classes of graphs of bounded treewidth.

A slightly more interesting example are complete graphs, which have a superlinear number of edges, but a linear-size DAG compression:

▶ **Example 2.4** (Cliques)**.** Let $C_n := (V,E)$ with $E = V \times V$ be the complete graph on n vertices. Note that $m = |E| = n^2$. A linear-size ($n+1$ to be precise) DAG compression for it is (V', A, E') with $V' = V \cup \{c\}$, where c is a fresh vertex, $A = \{(c,v) \mid v \in V\}$ contains an edge from c to every vertex of V, making all of them sinks, and $E' = \{(c,c)\}$ contains a single loop. Indeed, we then have $E = \bigcup_{(u',v') \in E'} C(u') \times C(v') = C(c) \times C(c) = V \times V$.

A more involved and interesting example are cographs, which are the natural "base class" to define twinwidth (which we will discuss in more detail later):

▶ **Example 2.5** (Cographs)**.** The class of *cographs* is defined inductively as follows: First, any single vertex is a cograph. Second, if G and H are cographs, so are their disjoint union and also their disjoint union with all edges between vertices in G and vertices in H added. This inductive definition can be used to obtain a linear-size ($5n-4$ to be precise) DAG compression of any cograph: Compressing single vertex graphs is trivial, so let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be cographs and let D_G and D_H be DAG compressions of sizes $5n_G - 4$ and $5n_H - 4$, respectively. We will ensure (and assume) that the cluster DAGs C_G and C_H are actually trees with roots r_G and r_H.

A DAG compression of the disjoint union is then obtained by taking the disjoint union of the two compressions, adding a new root r and adding the edges (r, r_G) and (r, r_H), that is, by considering $\bigl(V'_G \cup V'_H \cup \{r\}, A_G \cup A_H \cup \{(r,r_G),(r,r_H)\}, E'_G \cup E'_H\bigr)$ and always assuming that all vertex names are distinct. Note that the resulting size is $5n_G - 4 + 5n_G - 4 + 2 = 5n - 6 \le 5n - 4$.

The interesting case is to obtain a DAG compression of the disjoint union with all edges between G and H added. However, this is easy to achieve by taking the same construction and just adding the edges (r_G, r_H) and (r_H, r_G) to E' as this will cause $C(r_G) \times C(r_H) = V_G \times V_H$ and also the reversed edges $C(r_H) \times C(r_G) = V_H \times V_G$ to be added to E, exactly as needed. This adds two more edges to the size of the DAG compression, meaning that the size is $5n_G - 4 + 5n_G - 4 + 2 + 2 = 5n - 4$ as claimed.

Compressing Weighted Graphs and Arbitrary Structures. It is straightforward to extend the definition of a DAG compression to weighted graphs: Simply add a weight function $w' \colon E' \to \mathbb{N}$ that assigns weights to compressed edges. The obvious semantics is then that for $(u',v') \in E'$ all edges in $C(u') \times C(v')$ should have weight $w'(u',v')$. However, we then run into the problem that different weights may now be assigned to the same edge (u,v), namely when $(u,v) \in C(u'_1) \times C(v'_1)$ and also $(u,v) \in C(u'_2) \times C(v'_2)$ for some compressed edges (u'_1, v'_1) and (u'_2, v'_2) with $w'(u'_1, v'_1) \ne w'(u'_2, v'_2)$. We resolve this case by assigning the *minimum* weight to (u,v) of the weights of all compressed edge that uncompress to (u,v). Formally, we require that for a weighted graph (V,E,w) a weighted DAG compression is a tuple (V', A, E', w') such that (V', A, E') is a DAG compression of (V,E) and for all $e \in E$ we have $w(e) = \min_{(u',v') \in E', e \in C(u') \times C(v')} w'\bigl((u',v')\bigr)$.

As mentioned in the introduction, it is straightforward to define DAG compressions of graphs with multiple edge relations E_i by using multiple compression relations E'_i (but still using a single cluster DAG). It makes also sense to use DAGs to compress not only binary edge relations, but also unary relations (subsets of V, that is, colors): We can compress a color $X \subseteq V$ using a set $X' \subseteq V'$ such that $X = \bigcup_{x' \in X'} C(x')$, that is, by representing X as the union of some clusters described by the cluster graph. In the other direction, is it also possible to compress *ternary* relations $R \subseteq V \times V \times V$ using a relation $R' \subseteq V' \times V' \times V'$ such that $R = \bigcup_{(u',v',w') \in R'} C(u') \times C(v') \times C(w')$; and note that this potentially allows one to compress relations with $|R| = O(n^3)$ using DAG compressions of size $O(n)$. All told, DAG compressions can be used to compress arbitrary logical structures as well, but for simplicity, we restrict our attention to (weighted) graphs in the following.

Cluster Trees Versus Cluster DAGs. In all of the above examples, the cluster DAG was actually a tree. The following is an important example of a graph for which we appear to need a DAG to compress it to linear size (we believe that one can prove that a linear-size compression using trees is not possible, but are not aware of any simple proof for this claim):

▶ **Example 2.6** (Rook Graph). The *rook graph on n vertices,* where $n = s^2$ is the square of some integer $s = \sqrt{n}$, is a graph G with $V = \{1, \dots, s\}^2$ and with $\big((i,j),(k,l)\big) \in E$ iff $i = k$ or $j = l$, that is, if a rook could be moved from position (i,j) to position (k,l) in a chess game in a single move. Another way of viewing a rook graph is as an intertwined union of cliques: Every row is a clique and every column is a clique, but there are not other edges. Note that the rook graph has $(2s)s^2 = \Theta(n^{3/2})$ edges.

We can easily construct a linear-size DAG compression for the rook graph: Consider (V', A, E') with $V' = V \cup \{r_1, \dots, r_s\} \cup \{c_1, \dots, c_s\}$, so we add a *row vertex* r_i for each row and similarly a *column vertex* c_i for each column; we set $A = \big\{(r_i, (i,j)) \mid i,j \in \{1, \dots, s\}\big\} \cup \big\{(c_j, (i,j)) \mid i,j \in \{1, \dots, s\}\big\}$, that is, each row vertex and each column vertex is directly connected to all vertices of their row or column, respectively; and we set $E' = \{(c_i, c_i) \mid i \in \{1, \dots, s\}\} \cup \{(r_i, r_i) \mid i \in \{1, \dots, s\}\}$, that is, we add self-loops at all row and column vertices, resulting in cliques in E for each row and each column – exactly, what we are looking for. The total size of the described DAG compression is $|A| + |E'| = 2s^2 + 2s = 2n + 2\sqrt{n} = O(n)$.

There is a deeper reason why we can compress the rook graph so well using DAGs rather than trees: Using DAGs in compressions allows us to *implement the union operation on edge sets by uniting the DAG compressions.* The formal statement is the following:

▶ **Lemma 2.7.** *Let $G = (V, E_1 \cup E_2)$ and let $D_1 = (V_1', A_1, E_1')$ and $D_2 = (V_2', A_2, E_2')$ be DAG compressions of (V, E_1) and (V, E_2), respectively, that use distinct vertex sets for non-sink vertices, that is, $V_1' \cap V_2' \subseteq V$. Then $(V_1' \cup V_2', A_1 \cup A_2, E_1' \cup E_2')$ is a DAG compression of G whose size is at most the sum of the sizes of D_1 and D_2.*

Proof. Since $V_1' \cap V_2' \subseteq V$, the edges of the cluster DAGs A_1 and A_2 do not "interfere," that is, in the new compression DAG for any $v' \in V_1'$ the set $C(v')$ with respect to reachability in $A_1 \cup A_2$ is the same as $C(v')$ with respect to just A_1; and symmetrically for $v' \in V_2'$. This implies that for any compressed edge $(u', v') \in E_1' \cup E_2'$, the set $C(u') \times C(v')$ is the same as before. In particular, the union of all of these sets is exactly $E_1 \cup E_2$. The claim concerning the sizes follows directly from the construction. ◀

By the lemma, the rook graph can be compressed simply because it is the union of $2\sqrt{n}$ cliques, each having $\sqrt{n}$ vertices and, hence, allowing a DAG compression of size $1 + \sqrt{n}$ by Example 2.4; so the lemma tells us that a size of $2\sqrt{n}(1 + \sqrt{n}) = O(n)$ suffices for the union of all these cliques – no matter how they are intertwined.

2.2 Updating DAG Compressions

When defining a new data structure, a natural question is how difficult it is to update it. That is, suppose we have already constructed a DAG compression D of a graph G, with D being stored in memory while G is not stored directly, and we now wish to modify G by adding or deleting edges or vertices. How difficult is it to update D instead (without decompressing it)? In other words, given D, we wish to compute a DAG compression $\tilde{D}$ of $\tilde{G}$, where $\tilde{G}$ results from G by some small change.

Let us start with simple modifications that are easy to implement. First, we may wish to add an edge, meaning that $\tilde{G} = \bigl(V, E \cup \{(u, v)\}\bigr)$. It is then fairly simple to compute $\tilde{D}$ in this case: We can add the edge as a compressed edge, that is, let $\tilde{E}' = E' \cup \{(u, v)\}$. Note that the size increases only by one. Second, we may wish to add a new vertex. This turns out to be even simpler: Just add it to V', where it will become an isolated sink. This does not even change the size of the compression. Third, we may wish to delete an existing vertex v from V along with all adjacent edges in E. This is also simple to achieve: Simply delete v from V' and all its occurrences in A and in E'.

One operation is suspiciously missing: Deleting an edge (u, v) from E. It turns out that this can be a difficult operation to implement: If $(u, v) \in C(u') \times C(v')$ for several compressed edges $(u', v') \in E'$, we need to "break up" these compressed edges, meaning that we need to remove the compressed edge (u', v') and to then add new compressed edges that cover exactly the set $(C(u') \times C(v')) \setminus \{(u', v')\}$. It is currently unclear to us what the exact complexity of this operation is.

Another suspiciously missing aspect is the question of what happens when we have multiple edge additions in a row. Clearly, it is not optimal to simply add each edge as a compressed edge that only compresses itself: If we add all edges of, say, a cluster $C(v')$ and thereby making it a clique, we would like to end up with a DAG compression in which there is a single compressed edge (v', v') in E' to represent this clique. Undoubtedly, greedy heuristics exist for locally compressing sets of newly inserted edges, but finding a minimum-size DAG compression (V', A, E') for a given graph (V, E) appears to be a difficult problem. The following theorem shows that minimizing $|E'|$ is NP-complete and we conjecture that minimizing $|A| + |E'|$ (which is the more important question from a practical point of view) is also NP-complete:

▶ **Theorem 2.8.** *It is* NP*-complete to decide on input $G = (V, E)$ and a number k whether there is a* DAG *compression (V', A, E') of G with $|E'| \le k$.*

Proof. Reduce from the NP-complete problem of covering a bipartite graph with at most k complete bipartite graphs [19]. By definition, a DAG compression (V', A, E') of G with $|E'| \le k$ immediately yields a cover of E by at most k complete bipartite graphs; and given such a cover of size k, we can easily construct a cluster DAG such that for each complete bipartite graph $X \times Y$ in this cover there are vertices x' and y' with $C(x') = X$ and $C(y') = Y$, allowing us to put the edge (x', y') into E'. ◀

3 DAG Compression: Algorithms

Given a DAG compression D of some graph G, we wish to solve typical algorithmic problems on G, for instance, we would like to compute a topological ordering of G. The objective is, of course, to do so *without* "decompressing" the graph, that is, without storing the large graph G in memory. Rather, we would like to directly work on D and would like to have linear or quasi-linear runtimes in terms of the size of D.

At first sight, DAG compressions seem rather ill-suited for this purpose: Even deciding whether there is an edge between two given vertices $u, v \in V$ is not straightforward. Indeed, to answer this simple question using only $D = (V', A, E')$, we have to determine whether there is a compressed edge $(u', v') \in E'$ such that u is reachable from u' in A and v is reachable from v' in A. If A is a complex graph containing long paths, this is a nontrivial problem. Indeed, even very simple problems like determining the degree of a vertex are difficult if only D is given, as we may need to consider all vertices $v' \in V$ from which v is reachable – and this set may have linear size.

Nevertheless, it turns out that many problems involving *the whole graph* G can be solved in linear-time with respect to D. The core idea behind these algorithms is the construction of the *switching graph*, whose core property is that it is distance-equivalent to G.

Distance Equivalence and the Switching Graph. In order to solve BFS in G using only D, we first construct a new graph S that is *distance equivalent* to G, but has few edges.

▶ **Definition 3.1** (Distance Equivalence). *Let $G_1 = (V_1, E_1, w_1)$ and $G_2 = (V_2, E_2, w_2)$ be two weighted graphs. They are* distance equivalent *(on $V_1 \cap V_2$) if for all $u, v \in V_1 \cap V_2$ we have $d_{G_1}(u, v) = d_{G_2}(u, v)$.*

The key observation, to be formalized later, is that computing, say, a BFS ordering of the vertices in G_1 will also yield a BFS ordering of the vertices in V_2 in G_2 because the ordering in which vertices need to be visited depends on the distances.

Let us now define the switching graph of a DAG compression D and prove that it is distance equivalent to the uncompressed graph G.

▶ **Definition 3.2** (Switching Graph). *Let $D = (V', A, E', w')$ be a DAG compression of a weighted graph $G = (V, E, w)$. For each $v' \in V'$ let $\bar{v}'$ be a new vertex, except when $v' \in V$, in which case $\bar{v}' = v'$. The* switching graph S *of a DAG compression $D = (V', A, E', w')$ of a weighted graph $G = (V, E, w)$ is a weighted graph $S = (V'', E'', w'')$ such that*

1. *the vertex set V'' is the union of the three sets*
   - upper part $V_{\text{upper}} = \{v' \mid v' \in V' \setminus V\}$,
   - middle part $V_{\text{middle}} = \{v \mid v \in V\} = \{\bar{v} \mid v \in V\}$, *and*
   - lower part $V_{\text{lower}} = \{\bar{v}' \mid v' \in V' \setminus V\}$,
2. *the edge set E'' is the union of the three sets*
   - upper cluster edges $\{(u', v') \mid (u', v') \in A\}$ in the upper part,
   - lower cluster edge $\{(\bar{v}', \bar{u}') \mid (u', v') \in A\}$ in the lower part, *and*
   - switching edges $\{(\bar{u}', v') \mid (u', v') \in E'\}$,
3. *and the weight function $w'' \colon E'' \to \mathbb{N}$ with $w''((u', v')) = w''((\bar{u}', \bar{v}')) = 0$ for the cluster edges resulting from $(u', v') \in A$ and with $w''((\bar{u}', v')) = w'((u', v'))$ for the switching edges resulting from $(u', v') \in E'$.*

An example of a switching graph is depicted in Figure 2, where the weights in G are all 1 and, hence, the weights in S are either 0 (for cluster edges, depicted in gray) or 1 (for switching edges, shown in black). For further reference, we note a trivial observation:

▶ **Lemma 3.3.** *For very switching graph we have $|V''| \leq 2|V'|$ and $|E''| = 2|A| + |E'|$.*

Let us now prove the main property of switching graphs:

▶ **Theorem 3.4.** *Let S be the switching graph of a DAG compression D of G. Then S and G are distance equivalent.*

Proof. Let u and v be a pair of vertices in V.

First, consider a minimum-weight u-v-walk $(v_0, \dots, v_l)$ in G and let k be its weight. We will construct a u-v-walk in S of the same weight, starting at $v_0 = u$ and extending it for each $i \in \{1, \dots, l\}$ each time to v_i. For a given i, we must have $(v_{i-1}, v_i) \in E$ as we have a walk in G. Since D is a DAG compression of G, there must exist $(v'_i, v'_{i+1}) \in E'$ such that $v_{i-1} \in C(v'_{i-1})$ and $v_i \in C(v'_i)$ and $w((v_{i-1}, v_i)) = w'((v'_{i-1}, v'_i))$. Extend the new walk as follows: From $v_{i-1} = \bar{v}_{i-1}$ use (reversed) cluster edges to reach $\bar{v}'_{i-1}$ in the lower part (which must exist since $v_{i-1} \in C(v'_{i-1})$ means that v_{i-1} is reachable from v'_{i-1} using non-reversed cluster edges, so $\bar{v}'_{i-1}$ is reachable from $\bar{v}_{i-1}$ using reversed cluster edges), use the switching edge $(\bar{v}'_{i-1}, v'_i)$ to get to the upper part, and use cluster edges in the upper part to get to v_i. We only use exactly one switching edge during this extension of the new walk, meaning that the weight of the walk increases exactly by the weight of this edge. This immediately yields the claim concerning the total walk weight.

Second, consider a minimum-weight u-v-walk $(v''_0, \dots, v''_l)$ in S and let k be its weight. Since $u = v''_0$ and $v = v''_l$, we start and end in the middle part of V''. We cut the walk into subwalk $P_1, \dots, P_p$ of minimal lengths (but at least 1) such that each P_i starts and ends with a vertex in the middle part (that is, in V), while all other vertices are in the lower or in the upper part. Each subwalk (except for P_1) begins with the last vertex of the previous subwalk. As an example, the example 3-9-walk $(3, \overline{10}, 11, 5, \overline{12}, 14, 9)$ in Figure 2 would be cut into the subwalks $P_1 = (3, \overline{10}, 11, 5)$ and $P_2 = (5, \overline{12}, 14, 9)$ since V contains only the single digit numbers. Observe that the number of subwalks is exactly the number of positions $j \in \{1, \dots, l\}$ for which $v''_j \in V$ holds, that is, how often the walk crosses the middle part.

We claim that each P_i contains exactly one switching edge and all other edges are cluster edges. To see this, let $P_i = (p_1, \dots, p_z)$ and observe that only p_1 and p_z lie in the middle part by construction. From p_1, all non-switching edges point to a vertex in the lower part – and this is true also for all vertices in the lower part. Thus, up to the first switching edge on P_i, all edges are (reversed) lower cluster edges. Then, at some point, a switching edge $(\bar{p}', q') \in E''$ must be used for some $(p', q') \in E'$ since, otherwise, we could not exit the lower part (p_z is not in the lower part – and we also not allowed to just "rest" at p_1 since we must make at least one step as the length of all P_i is at least 1). Note that $\bar{p}'$ is reachable from p_1, meaning $p_1 \in C(p')$. By construction, the switching edge brings us to the upper part (or to the middle part, but then we stop and are done). In the upper part, we can only follow upper cluster edges until we reach the middle part; but then we stop one more, having reached p_z. This implies that $p_z \in C(q')$.

We see that each P_i consists of cluster edges (having weight 0) and a single switching edge $(\bar{p}', q') \in E''$ of some weight $w''((\bar{p}', q')) = w'((p', q'))$ for $(p', q') \in E'$. Furthermore, $(p_1, p_z) \in E$ must hold since $p_1 \in C(p')$ and $p_y \in C(q')$. All told, for each subwalk P_i starting at some vertex $p \in V$ and ending at a vertex $q \in V$, we see that there is an edge $(p, q) \in E$. Furthermore, the weight of this edge is at most the weight of the switching edge used in P_i. However, it also cannot be smaller than this weight: Otherwise, we could replace the walk by another walk from p to q in S of lesser weight (simple walk from p to the switching edge having this smaller weight and then walk to q). This shows that the minimal weight of a u-v-walk in G is k. ◀

Our first main theorem from the introduction, Theorem 1.1, is just a restatement of the above theorem.

▶ **Corollary 3.5.** *Given a* DAG *compression $D = (V', A, E')$ of a graph $G = (V, E)$, we can run a* BFS *in time $O(|V'| + |A| + |E''|)$.*

Proof. Run Dijkstra's algorithm [11] on the switching graph S of D, which has size at most $2|V'|$ vertices and $2|A| + |E'|$ edges by Lemma 3.3. Since all weights are 0 or 1, we can run the Dijkstra algorithm in time $O(|V'| + |A| + |E'|)$. ◀

▶ **Corollary 3.6.** *Given a* DAG *compression $D = (V', A, E')$ of a graph G, we can run a* DFS *in time $O(|V'| + |A| + |E''|)$.*

Proof. Modify Dijkstra's algorithm in Corollary 3.5 to extract the maximum instead of the minimum. ◀

Theorem 1.2 directly follows from Corollaries 3.5 and 3.6. When the graphs we study are weighted, as in Theorem 1.6, we can still run the Dijkstra algorithm, but the runtime is no longer linear in the size of the DAG compression, but of the order $O(s \log s)$, where $s = |V'| + |A| + |E'|$. We obtain Theorem 1.6 from the introduction.

4 DAG Compressions: The Link to Twinwidth

We have shown that several standard algorithms can be implemented in time linear in the size of the DAG compression of a graph, and we also saw examples of graphs such as cographs or the rook graph that allow us to compress graphs with $n^{1+\delta}$ edges for $\delta > 0$ to size $O(n)$. In the following we show that there is a large class of graphs for which a linear-size DAG compression is always possible: Graphs of bounded twinwidth. Linear-sized DAG compressions are tightly linked with bounded twinwidth, however, they do not capture the same class of graphs (as the rook graph shows). We first define twinwidth and afterwards show how we can build the DAG compression from a so-called contraction sequence. Readers familiar with twin-models [6] will note that the tree compressions developed in the following are very closely related to twin-models (the difference is mainly in the terminology). For simplicity, we only consider undirected graphs, that is, graphs with a symmetric edge relation.

Twinwidth and Contraction Sequences. Following Bonnet et al. it will be useful to consider *trigraphs* [4, 5], which are triples (V, B, W, R) such that R, B, and W partition the set of possible (non-loop) edges $V \times V \setminus \{(v, v) \mid v \in V\}$ into *red edges*, *black edges*, and *white edges*. The *red*, *black*, or *white degree* of a vertex is the number of its incoming (or, equivalently, outgoing) red, black, or white edges, respectively.

A *contraction* in a trigraph G merges two arbitrary vertices u and v into a single fresh vertex z, forming a new trigraph G' by removing u and v and coloring for each $x \in V \setminus \{u, v\}$ the edge between x and z (and also the backward edge as the graph is symmetric) as follows: If the edges between x and u and between x and v were both black, so is the x-z-edge; if the edges were both white, so is the x-z-edge; and in all other cases the x-z-edge becomes red.

A *contraction sequence* of a graph $G = (V, E)$ is a sequence $(G_n, G_{n-1}, \dots, G_1)$ of trigraphs G_i such that the first $G_n = \big(V, E, V \times V \setminus (E \cup \{(v, v) \mid v \in V\}), \emptyset\big)$ is the trigraph in which the edges in E are black and everything else is white and there are no red edges; the last graph G_1 is just a single vertex; and each G_i results from G_{i+1} through the contraction of two vertices u_{i+1} and v_{i+1} of G_{i+1} into a fresh vertex z_i. The *red degree* of a contraction sequence is the maximum red degree any vertex in any of the G_i has. The sequence is called a d-*contraction sequence* if its maximum red degree is d. Finally, the *twinwidth* $\operatorname{tww}(G)$ of G is the minimal d for which there is a d-contraction sequence of G.

An example of 1-contraction sequence of the cycle C_4 is show in Figure 3.

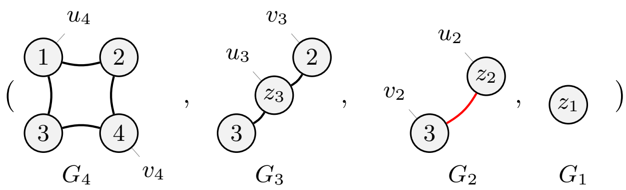


Figure 3 Example of a contraction sequence of the cycle C_4. In the first step, $u_4 = 1$ and $v_4 = 4$ are contracted to form the new vertex z_3 in G_3. The edge from z_3 to 2 is black since both the edges from u_4 to 2 and from v_4 to 2 were black (that is, present). Similarly, there is a black edge from z_3 to 3. In contrast, when $v_3 = 2$ and $u_3 = z_3$ are contracted to z_2 in G_2, the edge from z_2 to 3 is red since there was a (black) edge from u_3 to 3 in G_3, but a white (not present) edge from v_3 to 3. The sequence is a 1-contraction sequence since the maximum red degree of any vertex in the sequence is 1, proving that the twinwidth of C_4 is at most 1 (it is actually 0 since contracting 2 and 3 in G_3 rather than 2 and z_3 yields a 0-contraction sequence).

From Contraction Sequence to DAG Compression. Intuitively, contraction sequences "form clusters of vertices in a tree-like manner through contractions" and we can use this idea to build cluster DAGs from contraction sequences. Crucially, we can then insert compressed edges whenever a black edge is "about to finally disappear" and this will allow us to keep their number small (that is, linear in the number of vertices).

In detail, let us now assume that a fixed d-contraction sequence $(G_n, \dots, G_1)$ for a graph $G = (V, E)$ is given in which for each $i \in \{n, \dots, 2\}$ we contract u_i and v_i in the trigraph G_i into z_{i-1} in order to form the trigraph G_{i-1}. Our objective is to construct a DAG compression $D = (V', A, E')$ for G using the sequence such that $|A| + |E'| \le (3d + 4) \cdot n$. In particular, for every fixed d, the size of D is in $O(n)$.

The first step is to define the cluster DAG $C = (V', A)$. This is done as follows, see Figure 4 for an example:

▶ **Definition 4.1.** *The* cluster tree $C = (V', A)$ *of a contract sequence* $(G_n, \dots, G_1)$ *has* $V' = V \cup \{z_{n-1}, \dots, z_1\}$ *and for each* $i \in \{2, \dots, n\}$ *there are edges from* z_{i-1} *to both* u_i *and* v_i *to* A*, that is,* $A = \{(z_{i-1}, u_i) \mid i \in \{2, \dots, n\}\} \cup \{(z_{i-1}, v_i) \mid i \in \{2, \dots, n\}\}$.

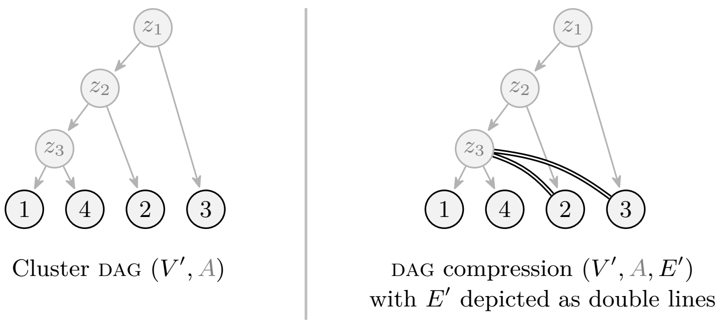


Figure 4 The cluster DAG resulting from the contraction sequence from Figure 3 and the DAG compression of C_4 resulting from it. The two compressed edges $(z_3, 2)$ and $(z_3, 3)$ are both added when G_3 is contracted to G_2, but for different reasons: In G_3 there is a black edge $(z_3, 2)$, but $(\alpha_3(z_3), \alpha_3(2)) = (z_2, z_2)$ is not a black edge in G_2 (there are no self-loops). In G_3 there is also a black edge $(z_3, 3)$, but while in G_2 there is an edge $(\alpha_3(z_3), \alpha_3(3)) = (z_2, 3)$, it is red.

A simple lemma will be useful in the following:

▶ **Lemma 4.2.** *Let (V', A) be the cluster tree of a contraction sequence $(G_n, \dots, G_1)$ of $G = (V, E)$ and let $x', y' \in V'$ be two different vertices in some G_i. Then there is a black edge between x' and y' in G_i if, and only if, $C(x') \times C(y') \subseteq E$.*

Proof. By induction. The start G_n is trivial. Suppose the claim holds for G_i, we need to show that it also holds for G_{i-1}, where u and v have been contracted to z. Consider two different vertices x' and y' in G_{i-1}. If neither of them is z, then the contraction does not change whether there is a black edge between them, so the induction hypothesis yields the claim. Since the vertices must be different, the only remaining case we need to consider is $x' \neq y' = z$. First, suppose that there is black edge (x', z) in G_{i-1}. By definition of contractions, the existence of this black edge implies that there are black edges (x', u) and (x, v) in G_i. By the induction hypothesis this yields that $C(x') \times C(u) \subseteq E$ and also $C(x') \times C(v) \subseteq E$. But, then, by construction we have $C(y') = C(z) = C(u) \cup C(v)$ and hence $C(x') \times C(y') \subseteq E$. Second, there is no black edge (x', z) in G_{i-1}. Then either (x', u) or (x, v) was not a black edge in G_i. By the induction hypothesis, either $C(x') \times C(u) \not\subseteq E$ or $C(x') \times C(v) \not\subseteq E$. Since we still have $C(y') = C(z) = C(u) \cup C(v)$, we get $C(x') \times C(y') \not\subseteq E$. ◀

The second step is to add as few compression edges as possible, that is, to keep E' small (while, of course, D is still a compression of G), by adding compression edges "as late as possible". In detail, for $i \in \{n, \dots, 2\}$ let $\alpha_i \colon V' \to V'$ be the function that maps both u_i and v_i to z_{i-1} and is the identity otherwise, so $\alpha_i(u_i) = \alpha_i(v_i) = z_{i-1}$ and $\alpha_i(v) = v$ for $v \in V' \setminus \{u_i, v_i\}$. In other words, α_i tells us "what became of a vertex v from G_i in G_{i-1}." We now add *a compression edge (u, v) to E' whenever (u, v) is a black edge in G_i but $(\alpha_i(u), \alpha_i(v))$ is no longer black or no longer present in G_{i-1}*. Formally:

▶ **Definition 4.3.** *The set of* compressed edges E' *of a contraction sequence $(G_n, \dots, G_1)$ is $E' = \big\{(u, v) \in V' \times V' \mid$ there is an $i \in \{2, \dots, n\}$ with $(u, v) \in B_{G_i}$, but $(\alpha_i(u), \alpha_i(v)) \notin B_{G_{i-1}}\big\}$.*

▶ **Theorem 4.4.** *For each (undirected, loop-free) graph G of twinwidth at most d there is a* DAG *compression $D = (V', A, E')$ with $|V'| = 2n - 1$, $|A| = 2n - 2$, and $|E'| \le 2 \cdot (2d + 1) \cdot (n - 1)$.*

Proof. Since G has twinwidth at most d, there is a d-contraction sequence $(G_n, \dots, G_1)$ for it. Consider the DAG compression $D = (V', A, E')$ where (V', A) is the cluster tree from Definition 4.1 for the contraction sequences and E' is the set of compressed edges from Definition 4.3 for the sequence.

The claim concerning the size of $|V'|$ follows trivially from Definition 4.1. For the size of $|A|$, just note that a binary tree on k vertices has $k - 1$ edges. For the size of E', we must count "how many black edges can disappear when G_i is contracted into G_{i-1}." First, if there is a black edge (u_i, v_i) in G_i, it will disappear, resulting in $(u_i, v_i) \in E'$. Second, if there is a vertex $v \in V' \setminus \{u_i, v_i\}$ such that (v, u_i) is black, but $(\alpha_i(v), \alpha_i(u_i)) = (v, z_{i-1})$ is red, we will add (v, u_i) to E' (and also (u_i, v), the symmetric edge, which is accounted for by the factor of 2 in the bound of the theorem). Likewise, if (v, v_i) is black, but (v, z_{i-1}) is red, we add (v, v_i) to E'. Crucially, *for any v, when at least one of the at most two black edges (v, u_i) or (v, v_i) is added to E', the edge (v, z_{i-1}) is red.* Since the maximum red degree of any vertex, including z_{i-1}, is d, there can be at most d red edges and hence at most $2d$ black edges may have disappeared. All told, from G_i to G_{i-1} we add at most $2 \cdot (2d + 1)$ compressed edges to E'.

It remains to argue that D is, indeed, a DAG compression of G. However, Lemma 4.2 immediately implies that all compressed edges we add to E' are "correct", that is, for every compressed edge $(u', v') \in E'$ we have $C(u') \times C(v') \subseteq E$. Furthermore, the lemma also implies that we do no "miss" any edges from E: Every edge of E is present in G' as a black edge and remains present as an element of some $C(u') \times C(v')$ until the black edge disappears – which is exactly the moment we add (u', v') to E'. Finally, G_1 is a single vertex and contains no edges, so all edges in E will be accounted for in E' at some point. ◀

By the above theorem, all graphs of twinwidth d admit a DAG compression (indeed, even a tree compression) of size $O(d \cdot n)$. However, *computing* the DAG compression of a graph of bounded twinwidth is a potentially difficult problem since it is already NP-hard to decide whether the twinwidth of a graph is 4 and, hence, it is also impossible to compute optimal contraction sequences in polynomial time unless $\mathrm{P} = \mathrm{NP}$. For these reasons, we can only hope to be able to compute the DAG compression of G when we are given a d-contraction sequence already as part of the input – and it is standard practice in the literature for algorithms working on graphs of twinwidth d to assume that this is the case.

However, one needs to be careful *how* the contraction sequence is represented in the input. Clearly, it makes little sense to assume that a string encoding the actual sequence $(G_n, G_{n-1}, \dots, G_1)$ is given as input – when G is dense, we wish to avoid explicitly keeping all edges of G_n in memory, let alone storing the whole sequence. Also, statements like "BFS can be solved in time $O(dn)$ for a graph G when a d-contraction sequence of G is given" are much less impressive if this presupposes that the input may take up $O(n^3)$ cells of memory. On the other hand, it is also not sufficient to just be given, say, for each $i \in \{n, \dots, 2\}$ the vertices u_i and v_i that get contracted to z_{i-1}: We then miss the information which black edges got removed.

What we really need is, in addition to the contraction pairs, for each $i \in \{n, \dots, 2\}$ for each red edge (v, z_{i-1}) the color of the edges (v, u_i) and (v, v_i): We can then reconstruct which black edges were lost from G_i to G_{i-1}. The following definition and theorem make these observations explicit:

▶ **Definition 4.5.** *Let $(G_n, \dots, G_1)$ be a contraction sequence such that in G_i we contract u_i and v_i into z_{i-1} to get G_{i-1}. Define the* color recording sequence *of the contraction sequence as a sequence of tuples $(t_n, t_{n-1}, \dots, t_2)$ such that each t_i contains the following:*
1. *The vertices u_i, v_i, z_{i-1}.*
2. *The color of the edge (u_i, v_i) in G_i.*
3. *For each vertex v such that (v, z_{i-1}) is a red edge in G_{i-1}, the colors of the edges (v, u_i) and of (v, v_i) in G_i.*

Observe that for a d-contraction sequence each tuple t_i in the color recording sequence contains at most $2d + 4$ vertices and, hence, the whole sequence can be stored using $O(d \cdot n)$ words of memory.

▶ **Theorem 4.6.** *There is an algorithm that gets a color recording sequence of a d-contraction sequences of a graph G as input and outputs in time $O(d \cdot n)$ a DAG compression D of G of size $O(d \cdot n)$.*

Proof. Having a look at the definitions of cluster trees (Definition 4.1) and of how the compressed edges E' are derived from a contraction sequence (Definition 4.3), we immediately see that the color recording sequence contains exactly the information needed to output (V', A, E') in time linear in the size of this output. ◀

Of course, the above theorem and definition beg the question of where the "color recording sequences" might come from. Firstly, it may be the case that we *do* have access (perhaps only during a preprocessing phase) to the graphs G_i. In this case, assuming that they are stored using standard data structures that combine the advantages of an adjacency matrix and list [1] and also assuming that we are told which vertices get contracted in each step, we can easily compute the color recording sequence by iterating over the red edges incident to each z_{i-1}. Secondly, the graph G and the contraction sequence may be the output of some algorithmic process. In this case, one needs to adapt the output or the process so that the color recording sequence gets output.

Graphs with Large Twinwidth and Linear-Size DAG Compressions. The results in this section suggest a tight link between twinwidth and linear-size DAG compressions: When G has twinwidth d, then G also has a size-$O(dn)$ DAG compression. Indeed, it even has a size-$O(dn)$ tree compression, that is, a DAG compression where $C = (V', A)$ is a tree. This raises the question of whether, perhaps, the reverse is also true: It is true that all graphs having a, say, linear-size tree compression, also have low twinwidth? The answer to this is negative:

▶ **Theorem 4.7.** *There is a sequence of graphs $(G_1, G_2, \dots)$ such that G_d has twinwidth at least d, but each $G_d = (V_d, E_d)$ admits a tree compression of size at most $|V_d|$.*

Proof. For each d, we can construct a graph H_d that has twinwidth at least d: For instance, the rook graph on n vertices from Example 2.6 has twinwidth at least $\sqrt{n}$ since contracting any two different vertices immediately yields a vertex with $\sqrt{n}$ incident red edges.

Let G_d be the graph H_d to which we add $n_d^2 - n_d$ isolated vertices, where n_d is the number of vertices of H_d. Then G_d has n_d^2 vertices and also twinwidth at least d since adding isolated vertices does not change the twinwidth. The number of edges in G_d equals that of H_d, so it can be at most n_d^2. This shows that $|E_d| \le n_d^2 = |V_d|$; in other words, the graph has a linear number of edges. As shown in Example 2.3 we can the "compress" it by simply doing nothing to get a size-$|V_d|$ compression. ◀

5 Conclusion

In this paper, we presented a new data structure, the DAG *compression,* that represents graphs by storing complete bipartite subgraphs as single compression edges between vertices that represent clusters of vertices. We showed that some update operations are possible on these compressions, but further research is needed to better understand them. A crucial property was that the DAG compression of a graph gives rise to another graph, which we called the *switching graph*, that is distance-equivalent to the original graph and is only twice the size of the compressed graph.

When the size of a DAG compression is linearly bounded by the number of vertices in the original graphs, the compression gives rise to a new framework for graph algorithms with running times that are independent of the number of edges in the input: We can run breadth- and depth-first search in time $O(n)$ where n is the number of vertices in the input, if we have access to a linear-size DAG compression. Moreover, extending the definition to weighted graphs, we can run Dijkstra's algorithm in time $O(n \log n)$ on such graphs. We believe that further algorithms that work directly on DAG compressions rather than on the original graphs are possible.

We also showed that all graphs of bounded twinwidth admit a linear-size DAG compression. The reverse, however, is not true. A natural research avenue would be to extend this result to further graph classes: Is it true that all graphs of, say, bounded flip-width admit a linear-size DAG compression?

References

1 Faisal N. Abu-Khzam, Michael A. Langston, Amer E. Mouawad, and Clinton P. Nolan. A hybrid graph representation for recursive backtracking algorithms. In *Proceedings of the 4th International Workshop on Frontiers in Algorithmics (FAW 2010)*, volume 6213 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2010. doi:10.1007/978-3-642-14553-7_15.

2 Jungho Ahn, Debsoumya Chakraborti, Kevin Hendrey, Donggyu Kim, and Sang il Oum. Twin-width of random graphs. Technical Report arXiv:2212.07880, arXiv, 2022. doi:10.48550/arXiv.2212.07880.

3 Pierre Bergé, Èdouard Bonnet, and Hugues Déprés. Deciding twin-width at most 4 is NP-complete. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *Proceedings of the 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics*, pages 18:1–18:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICALP.2022.18.

4 Édouard Bonnet, Colin Geniet, Eun J. Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width III: Max independent set and coloring. In *Proceedings of the 48th International Colloquium on Automata, Languages and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics*, pages 35:1–35:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.35.

5 Édouard Bonnet, Eun J. Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: Tractable FO model checking. In *Proceedings of the 61st Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 601–612. IEEE, 2020. doi:10.1109/FOCS46700.2020.00062.

6 Édouard Bonnet, Jaroslav Nesetril, Patrice O. de Mendez, Sebastian Siebertz, and Stephan Thomassé. Twin-width and permutations. In *Proceedings of the European Conference on Combinatorics, Graph Theory and Applications 2023 (EUROCOMB 2023)*. Masaryk University, Brno, Czech Republic, 2023. doi:10.5817/CZ.MUNI.EUROCOMB23-022.

7 Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topic-specific web resource discovery. *International Journal of Computer and Telecommunications Networking*, 31(11-16):1623–1640, 1999. doi:10.1016/S1389-1286(99)00052-3.

8 Chris J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970. doi:10.1145/362790.362798.

9 Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. *International Journal of Computer and Telecommunications Networking*, 30(1–7):161–172, 1998. doi:10.1016/S0169-7552(98)00108-1.

10 Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

11 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

12 Jeff Erickson. *Algorithms*. Erickson, 2019.

13 Robert Ganian, Filip Pokrývka, André Schidler, Kirill Simonov, and Stefan Szeider. Weighted model counting with twin-width. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *Leibniz International Proceedings in Informatics*, pages 15:1–15:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.15.

14 Michel Habib and Christophe Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 145(2):183–197, 2005. doi:10.1016/J.DAM.2004.01.011.

15 Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Journal of Artificial Intelligence*, 27(1):97–109, 1985. doi:10.1016/0004-3702(85)90084-0.

16 Stefan Kratsch, Florian Nelles, and Alexandre Simon. On triangle counting parameterized by twin-width. Technical Report arXiv:2202.06708, arXiv, 2022. doi:10.48550/arXiv.2202.06708.

17 David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, 1983. doi:10.1145/2402.322385.

18 Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*, pages 419–432. Association for Computing Machinery, 2008. doi:10.1145/1376616.1376661.

19 James Orlin. Contentment in graph theory: Covering graphs with cliques. *Indagationes Mathematicae (Proceedings)*, 80(5):406–424, 1977. doi:10.1016/1385-7258(77)90055-5.

20 Judea Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

21 André Schidler and Stefan Szeider. A SAT approach to twin-width. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, (ALENEX 2022)*, pages 67–77. Society for Industrial and Applied Mathematics, 2022. doi:10.1137/1.9781611977042.6.

22 Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981. doi:10.1016/0898-1221(81)90008-0.

23 Robert E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6:171–185, 1976. doi:10.1007/BF00268499.

24 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.

25 Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*, pages 567–580. Association for Computing Machinery, 2008. doi:10.1145/1376616.1376675.

26 Hannu Toivonen, Fang Zhou, Aleksi Hartikainen, and Atte Hinkka. Compression of weighted graphs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2011)*, pages 965–973. Association for Computing Machinery, 2011. doi:10.1145/2020408.2020566.

27 Ning Zhang, Yuanyuan Tian, and Jignesh M. Patel. Discovery-driven graph summarization. In *Proceedings of the 26th International Conference on Data Engineering (ICDE 2010)*, pages 880–891. IEEE, 2010. doi:10.1109/ICDE.2010.5447830.

28 Édouard Bonnet, Eun J. Kim, Amadeus Reinald, Stéphan Thomassé, and Rémi Watrigant. Twin-width and polynomial kernels. In Petr A. Golovach and Meirav Zehavi, editors, *Proceedings of the 16th International Symposium on Parameterized and Exact Computation, (IPEC 2021)*, volume 214 of *Leibniz International Proceedings in Informatics*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.IPEC.2021.10.