To Store or Not to Store: a graph theoretical approach for Dataset Versioning

Anxin (Bob) Guo[*]
anxinbguo@gmail.com
Northwestern University
Evanston, Illinois, USA

Jingwei (Sofia) Li[*]
jingweisofili@gmail.com
Northwestern University
Evanston, Illinois, USA

Pattara Sukprasert
pattara@u.northwestern.edu
Northwestern University
Evanston, Illinois, USA

Samir Khuller
samir.khuller@northwestern.edu
Northwestern University
Evanston, Illinois, USA

Amol Deshpande
amol@umd.edu
University of Maryland
College Park, Maryland, USA

Koyel Mukherjee
komukher@adobe.com
Adobe Research
India

ABSTRACT

Our work is motivated by the study of the *Dataset Versioning* problem: given a graph, where vertices denote different versions and edges capture edit operations to derive one version from another, an efficient storage scheme needs to make decisions as to which versions to store and which to reconstruct on demand using the stored versions and edit operations. In one central variant, which we call MinSum Retrieval (MSR), the goal is to minimize the sum of retrieval costs given that the storage cost is bounded. This problem arises in every collaborative tool, e.g., version control for source code, and/or intermediate data sets constructed during data analysis pipelines. The problem we study (and its variants) was originally formulated in the pioneering paper by Bhattacherjee et al. [VLDB' 15], which modeled this as a graph problem and developed a reasonably fast and efficient heuristic called Local Move Greedy (LMG). While being a fundamental problem encountered in practical systems, there is no research that studies the approximability hardness of these questions.

We first show that the best known heuristic, namely LMG, can be arbitrarily bad in some situations. In fact, it is hard to get $o(n)$-approximation for MSR on general graphs even if we relax the storage constraints by $O(\log n)$ times. Similar hardness results are shown for other variants.

Motivated by the fact that the graphs arising in practice from typical edit operations have some nice underlying properties, we propose poly-time approximation algorithms that work very well when the given graphs are tree-like.

As version graphs typically have low treewidth, we develop new algorithms for bounded treewidth graphs.

On the experimental side, we propose two new heuristics. First, we extend LMG, the best-known heuristic, by considering more potential "moves" and come up with a new heuristic named LMG-All. The run time of LMG-All is comparable to LMG, while it consistently performs better than LMG across a wide variety of datasets, i.e., version graphs. Second, we utilize our algorithm for tree instances on the minimum-storage arborescence of an instance, yielding an algorithm that is qualitatively better than LMG and LMG-All, but with a worse running time.

Apart from MinSum Retrieval, we show similar results for all other variants defined by Bhattacherjee et al. [VLDB' 15].

[*]Both authors contributed equally to this research.

PVLDB Reference Format:
Anxin (Bob) Guo, Jingwei (Sofia) Li, Pattara Sukprasert, Samir Khuller, Amol Deshpande, and Koyel Mukherjee. To Store or Not to Store. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Sooooffia/Graph-Versioning.

1 INTRODUCTION

Tremendous amount of data is produced daily due to the increasing usage of online collaboration tools for data storage and management: multiple users might collaborate to produce many versions of a raw data set. The management of all these versions, however, has become increasingly challenging in large enterprises. When we have thousands of versions, each of several terabytes, then storing all versions is extremely costly and wasteful. Reducing data storage and data management costs is a major concern for enterprises [49].

Important not just in online collaborative settings, dataset versioning is also a key concern for enterprise data lakes as well, that are managing huge volumes of customer data [51]. Often, existing versions of huge tabular datasets might require a few records (or rows) to be modified (for example, product catalogs), thus resulting in a new version for each such modification. This becomes challenging at the terabyte and petabyte scale and storing all of the versions can incur a huge storage and data management cost for the enterprise. Additionally, dataset versioning is a concern for Data Science and Machine learning (and Deep Learning) pipelines as well, as several versions of the data can get generated by the applications of simple transformations on existing data for training and insight generation purposes, thereby increasing the storage and management costs. It is no surprise therefore that data version control is emerging as one of the hot areas in industry [1–6], and even popular cloud solution providers like Databricks are now

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

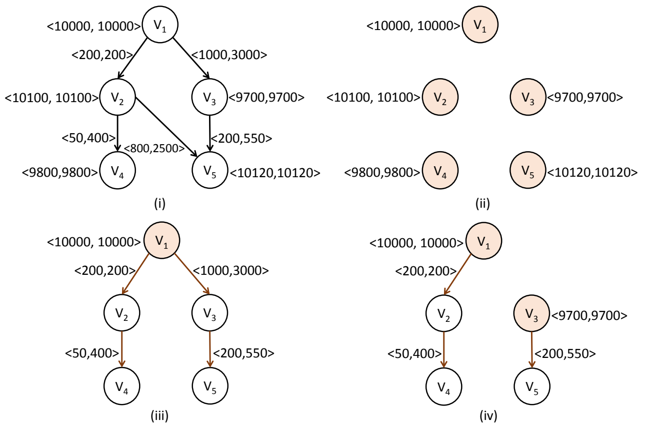

Figure 1: (i) A version graph over 5 versions – annotation ⟨a, b⟩ indicates a storage cost of a and a retrieval cost of b; (ii, iii, iv) three possible storage graphs. The figure is taken from Bhattacherjee et al. [11]

capturing data lineage information that can help in effective data version management [53].

In a pioneering paper, Bhattacherjee et al. [11] proposed an innovative model capturing the trade-off between *storage* cost and *retrieval* (recreation) cost. The common use case of their model includes version control (e.g., git, mercurial, svn), collaborative data science and analysis, and sharing data sets and intermediate results among data pipelines. The problem studied by the authors can be defined as follows. Given versions and a subset of the *"deltas"* between them (directed edges connecting versions), find a compact representation that minimizes the overall storage as well as the retrieval costs of the versions. This involves a decision for each version – either we *materialize* it, (store it explicitly) or we rely on the edit operations to retrieve the version from another materialized version when necessary. The downside of the latter is that to retrieve a version that was not materialized, we will have to incur a computational overhead as well as a delay while the user waits.

There are some follow-up works [22, 36, 61]. However, those works either formulate new problems in different use cases [22, 36, 47] or implement a system incorporating the feature to store specific versions and deltas [36, 54, 58]. We will discuss this in more detail in Section 2.3.

Figure 1, taken from Bhattacherjee et al. [11], illustrates the central point through different storage options. Here, vertices represent versions and edges represent deltas between versions. Abstractly, versions here can represent sourcecodes (e.g., git), intermediate datasets (on data pipelines), databases with snapshots, temporal graphs, etc. (i) shows the input graph, with annotated storage and retrieval costs (for an edge, the retrieval cost indicates the cost to reconstruct the target version given the source version). If the storage size is not a concern, we should store all versions as in (ii). For (iii) and (iv), it is clear that, by storing v_3, we shorten the retrieval times of v_3 and v_5.

To optimize the instance in Figure 1, one can think minimizing storage. In this case, the problem turns into an instance of Minimum Spanning Tree. In another case where the goal is to minimize retrieval cost, the problem turns into an instance of Shortest Path Tree. However, optimizing a single objective leave a lot to be desired.

This retrieval/storage trade-off leads to the combinatorial problem of minimizing one type of cost, given a constraint on the other. There are variations of our objective function as well: retrieval cost of a solution can be measured by either the maximum or total (or equivalently average) retrieval cost of files. This yields four different optimization problems (Problems 3-6 in Table 1).

	Problem Name	Storage Cost	Retrieval Cost
Prob. 1	Min Spanning Tree	min	$\mathcal{R}(v) < \infty,\ \forall v$
Prob. 2	Shortest Path Tree	$< \infty$	$\min\{\max_v R(v)\}$
Prob. 3	MinSum Retrieval (MSR)	$\leq \mathcal{S}$	$\min\{\sum_v R(v)\}$
Prob. 4	MinMax Retrieval (MMR)	$\leq \mathcal{S}$	$\min\{\max_v R(v)\}$
Prob. 5	BoundedSum Retrieval (BSR)	min	$\sum_v R(v) \leq \mathcal{R}$
Prob. 6	BoundedMax Retrieval (BMR)	min	$\max_v R(v) \leq \mathcal{R}$

Table 1: Problems 1-6

Fundamentally, we can think of the problem as a multi-root arborescence problem in a directed graph. A single-root arborescence in a directed (weighted) graph $G = (V, E)$ is a rooted tree such that every node is reachable from the root, and efficient algorithms for finding a minimum weight arborescence are known.

In our multi-root version, we can imagine that nodes have labels and our goal is to select a subset of nodes (multiple roots) and every node not selected should have a directed path from one of the roots to it. Edges may have multiple costs associated with them, relating to storage and computation (reconstruction) costs. The total sum of the node label values and edge costs directly contributes to the storage cost, and the path length to nodes, directly determines the delay to materialize those versions.

1.1 Our Contributions

We provide the first set of provable *inapproximability results* and *approximation algorithms* for the aforementioned optimization problems that trade-off between retrieval and storage costs from different angles.

Hardness. Table 2 summarizes all hardness results in this paper. Notably, it's impossible to approximate any of the problems within a constant factor on general directed graphs, with MSR being especially hard. This also motivated the consideration for special graph classes.

Tree Algorithms. We propose algorithms that work well when the given graph is a tree (even this special case is not trivial, as we will see). Specifically, for any graph whose underlying undirected graph is a tree , we show that BMR can be solved exactly and efficiently, and that there exists a $(1 + \epsilon)$-approximation algorithm for MSR.

Inspired by these algorithms on trees, we also proposed new heuristics on general graphs. Compared to the best heuristics in Bhattacherjee et al. [11], we improve the MSR solution by several orders of magnitude and the BMR solution by up to 50%.

Problem	Graph type	Assumptions	Inapproximability
MSR	arborescence	Triangle inequality Single weight[1]	1
	undirected		$1 + \frac{1}{e} - \epsilon$
	general		$\Omega(n)^2$
MMR	undirected		$2 - \epsilon$
	general		$\log^* n - \omega(1)$
BSR	arborescence		1
	undirected		$(\frac{1}{2} - \epsilon) \log n$
BMR	undirected		$(1 - \epsilon) \log n$

Table 2: Hardness results.

Graphs	Problems	Algorithm	Approx.	Run time
General Digraph	MSR	LMG-All	heuristic	close to LMG
Bounded Treewidth	MSR & MMR	DP-BTW	$1 + \epsilon$	$\text{poly}(n, \frac{1}{\epsilon})$
	BSR & BMR		$(1, 1 + \epsilon)$	
Bidirectional Tree	MMR	DP-BMR	exact	$n^2 \log \mathcal{R}_{max}$
	BMR			n^2

Table 3: Algorithms Summary. Here, $\mathcal{R}_{max}$ is defined to be the maximum retrieval cost between any pair of vertices in the tree.

Bounded Treewidth Algorithms. We extend our approximation algorithms for trees to graphs that are close to trees, as measured by a property called *treewidth* [9]. Our extensions to bounded treewidth graphs give $(1 + \epsilon)$-approximation for MSR and MMR, as well as $(1, 1 + \epsilon)$ bi-criteria approximation for BSR and BMR. One motivation behind our attention on bounded treewidth graph is that *Series–parallel graphs*[3], which are very similar to version graphs derived from repositories, has bounded treewidth. In fact, a graph has treewidth at most two if and only if every biconnected component is a series-parallel graph [13]. To confirm our hypothesis, we measure treewidths from various repositories. It is the case that the average treewidth is 3, while the highest treewidth among graphs we measured is 6.

New Heuristic. Additionally for MSR, we show that LMG, the algorithm proposed in [11], may perform arbitrarily poorly. This behavior is in line with the hardness results and is confirmed by experiments. On the other hand, we propose a slight modification named "LMG-All"(LMGA) which dominates the performance of LMG (by up to around 100 times in certain cases), while exhibiting decent run time on sparse graphs.

Inspired by our algorithms on trees, we also propose two new dynamic programming (DP) heuristics for MSR and BMR respectively. Both algorithms perform extremely well in almost all experiments, even when the input graph is not tree-like.

Due to space constraints, we omit most of the proofs; those can be found in the full version of the paper[4]. The full version will be made available on arXiv once the paper is published.

[1]Both are assumptions in previous work [11] that simplify the problems. In other words, our hardness results apply even when the weights r and s are equal on the edges, and when the weight satisfies the triangle inequality.
[2]This is true even if we relax $\mathcal{S}$ by $O(\log n)$.
[3]See, e.g., Eppstein [23] for formal definition.
[4]https://github.com/Sooooffia/Graph-Versioning/blob/main/Full-Paper.pdf

2 PRELIMINARIES

In this section, the definition of the problems, notations, simplifications, and assumptions will be formally introduced.

2.1 Problem Setting

In the problems we study, we are given a directed graph $G = (V, E)$. The given graph is a *version graph* where vertices represent *versions* and edges capture *"delta"* between versions. More precisely, every edge $e = (u, v)$ is associated with two weight functions: storage cost s_e and retrieval cost r_e.[5] It takes r_e time to retrieve v given that we have retrieved u. The cost of storing (materializing) e is s_e, and the cost of storing v is s_v. Since there is usually a smallest unit of retrieval/storage cost in real world, we will work with nonnegative integers, that is, $s_e, r_e \in \mathbb{N}$ for all $e \in E$.

In order to retrieve a version v from a materialized version u, there must be some path $P\{(u_{i-1}, u_i)\}_{i=1}^{n}$ with $u_0 = u, u_n = v$, such that all edges along this path are stored. In such cases, we say that v can be retrieved from u with retrieval cost $\sum_{i=1}^{n} r_{(u_{i-1}, u_i)}$. In the rest of the paper, we say v is "retrieved from u" if u is in the path to retrieve v, and v is "retrieved from materialized u" if in addition u is materialized.

The general optimization goal is to select a set of versions $M \subseteq V$ and a set of edges $F \subseteq E$ of **small** size (w.r.t. storage cost s) such that for each $v \in V \setminus M$, the length of shortest path in F (w.r.t. retrieval cost r) from any node $m \in M$ to v is optimized (different versions of the problem optimize different measures). We denote the cost of this shortest path as $R(v)$.

In some previous works, an **auxiliary root** is added to the graph to simplify the problem. Though not used in our approximation algorithms, this is an important simplification for defining problems and for many previous heuristics. We briefly explain the concept here: instead of finding both set of versions and set of edges, we could simplify the problem by adding an auxiliary root v_{aux} to the graph and let edges in the form (v_{aux}, v) capture the storage cost of storing v explicitly.

More precisely, we generate a graph $G_{aux} = (V_{aux} = V \cup v_{aux}, E_{aux} = E \cup E')$ from $G = (V, E)$ where $E' = \{(v_{aux}, v) \mid v \in V\}$. Each $e = (v_{aux}, v) \in E_{aux}$ has $s_e = s_v$ and $r_e = 0$. It is straightforward to see that any solution $M \subseteq V, F \subseteq E$ has a 1-to-1 correspondence with a directed spanning tree rooted at v_{aux}. In particular, problems 1 and 2 reduce to familiar problems on G_{aux} (see below).

2.2 Problem Definition

Different problems are formulated based on different optimization goals. Recall that, after simplification, we want to select a set of edges $F \subseteq E_{aux}$ such that $H = (V_{aux}, F)$ is an arborescence rooted at v_{aux}. We let $s(H) = \sum_{e \in F} s_e$ be the total storage cost. We also let $R(H) = \sum_{v \in V_{aux}} R(v)$ be the total retrieval cost.

Since the two objectives are negatively correlated, and since we want to capture both aspects, one natural way is to constrain one objective and optimize the other objective. The following optimization goals were originally defined in Bhattacherjee et al. [11] though we might use different names for brevity. See Table 1 for the 6 problem definitions.

[5]We may use $s_{u,v}$ in place of s_e and $r_{u,v}$ in place of r_e.

Since the first two problems are well studied, we do not discuss them further. In a way, MSR and BSR (MMR and BMR, resp.), are closely related. If we have an algorithm for MSR (MMR, resp.), we can turn it into an algorithm for BSR (BMR, resp.) by binary-searching over $\mathcal{S}$. The same is true for the other direction. Hence, to ease the representation, we will foucs only on two problems: MinSum Retrieval (MSR) and BoundedMax Retrieval (BMR).

2.3 Related Works

2.3.1 ***Theory.*** There has been little theoretical analysis on the exact problems we study. The optimization problems are first formalized in Bhattacherjee et al. [11], which also compared the effectiveness of several proposed heuristics on both real-world and synthetic data. They defined six variants of the problem, two of which are polynomial-time solvable, and the other four are NP-hard (see Section 2.2). Zhang et al. [61] followed-up by considering a new objective that's a weighted sum of objectives in MSR and MMR. They also modified the heuristics to fit this objective. There are similar concepts, including *Light Approximate Shortest-path Tree (LAST)* [41] and *Shallow-light Tree (SLT)* [32, 35, 40, 44, 48, 52]. However, this line of work focuses mainly on undirected graphs and their algorithms don't generalize to the directed case. Among the two problems mentioned, SLT is closely related to MMR and BMR. Here, the goal is to find a tree that is **light** (minimize weight) and **shallow** (bounded depth). To the best of our knowledge, there are only two works that give approximation algorithms for directed shallow-light trees. Chimani and Spoerhase [21] gives a bi-criteria $(1+\epsilon, n^{\epsilon})$-approximation algorithm that runs in polynomial-time. Their run-time analysis is quite complicated, but it is at least $n^{O(1/\epsilon)}$. Recently, Ghuge and Nagarajan [30] showed that a problem called "submodular tree orienteering" has a $O(\frac{\log n}{\log\log n})$ approximation algorithm that runs in quasi-polynomial time. In this problem, we want to find a directed tree T rooted at r such that $s(T) \leq \mathcal{S}$ and maximize $f(V(T))$ where the objective function f is a submodular function. The authors also extended their algorithm so that it works when both **retrieval costs** and **storage costs** are constrained. Their algorithm can be adapted into $O(\frac{\log^2 n}{\log\log n})$-approximation for MMR and BMR where the approximation part is on the storage cost. For MSR and BMR, their algorithm gives $\left(O(\frac{\log^2 n}{\log\log n}), O(\frac{\log^2 n}{\log\log n})\right)$-approximation. The idea is to run their algorithm for many rounds, where the objective of each round is to **cover as many nodes as possible**. We also note that our assumptions, namely, triangle inequality and integral weights are also used in their paper [30].

2.3.2 ***Systems.*** To implement a system captured by our problems, components spanning multiple lines of works are required. For example, to get a graph structure, one has to keep track of history of changes. This is related to the topic of data provenance [18, 56]. Given a graph structure, the question of modeling "deltas" is also of interest. There is a line of work dedicated to studying how to implement `diff` algorithms in different contexts [19, 37, 45, 57, 59].

In the case where we have more flexibility, one may think of creating deltas from different versions without much of the change history. However, computing all possible deltas is too wasteful, hence it is necessary to utilize other approaches to identify similar versions/datasets. Such line of work is known in the literature as dataset discovery or dataset similarlity [14, 16, 25, 38, 51].

After the work of Bhattacherjee et al. [11], there are several followup works that implemented systems with a feature that saves only selected versions to reduce redundancy. There are works that focus on version control for relational databases [10, 17, 20, 36, 46, 54, 55, 58] and works that focus on graph snapshots [42, 47, 60]. However, since their focuse was on designing full-fledged systems, the algorithms proposed for these systems are rather simple heuristics, without rigorous theoretical results. Here is a non-exhaustive list of examples. *OrpheusDB* [36] tackled a similar problem but was designed specifically for relational databases. *Forkbase* [58] is a version control system for blockchain-like instances with built-in fork semantics. *Pensieve* [60] is a system designed specifically for storing graphs. Derakhshan et al. [22] formulated a more generalized problem, which includes time intervals. However, since they deal with data science & machine learning pipelines, they only consider instances where the underlying graphs are directed acyclic.

2.3.3 ***Usecases.*** In a version control system such as git, our problem is similar to what `git pack` command aims to do.[6] The original heuristic for `git pack`, as described in an IRC log, is to sort objects in particular order and only create deltas between objects in the same window.[7] It is shown in Bhattacherjee et al. [11] that git's heuristic does not work well compared to other methods.[8] For svn, the most recent version and deltas to the past versions are stored [50]. Other existing data version management systems include [2–6], which offer git-like capabilities suited for different use cases, such as data science pipelines in enterprise setting, machine learning-focused, data lake storage, graph visualization, etc.

3 HARDNESS RESULTS

We hereby list the main hardness (inapproximability) results of the two problems.

3.1 Heuristics can be Arbitrarily Bad

First, we consider the approximation factor of the best heuristic for MSR in [11], Local Move Greedy (LMG). The gist of this algorithm is to start with the arborescence that minimizes the storage cost, and iteratively materialize a version that most efficiently reduces retrieval cost per unit storage. In other words, in each step, a version is materialized with maximum ρ where $\rho = \frac{\text{reduction in total of retrieval costs}}{\text{increase in storage cost}}$.

Note also that we work with the modified graph G_{aux} with the auxiliary root, as defined in Section 2.1. Here we show that, even on simple instances, LMG could perform poorly as an approximation algorithm.

Theorem 3.1. *LMG has an arbitrarily bad approximation factor for MinSum Retrieval, even under the following assumptions: (1) G is a directed path; (2) there is a single weight function; and (3) triangle inequality holds.*

[6]https://www.git-scm.com/docs/git-pack-objects

[7]https://github.com/git/git/blob/master/Documentation/technical/pack-heuristics.txt

[8]There is a blog post that further discusses the point: http://www.cs.umd.edu/~amol/DBGroup/2015/06/26/datahub.html

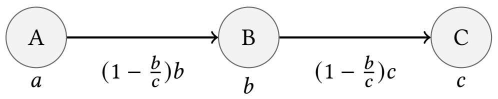

Figure 2: An adversarial example for LMG.

Proof. Consider the following chain of three nodes; the storage costs for nodes and the storage/retrieval costs for edges are labeled in Figure 2 (let a be large and $\epsilon = b/c$ be close to 0). To save space, we do not show v_{aux} but only the nodes of the version graph.

It's easy to check that triangle inequality holds on this graph.

In the first step of LMG, the minimum storage solution of the graph is $\{A, (A,B), (B,C)\}$ with storage cost $a + (1-\epsilon)b + (1-\epsilon)c$.

Next, in the greedy step, two options are available: (1). Choosing B and delete (A,B): $\rho_1 = \frac{2(1-\epsilon)b}{\epsilon b} = \frac{2}{\epsilon} - 1$; (2). Choosing C and delete (B,C): $\rho = \frac{(1-\epsilon)b+(1-\epsilon)c}{\epsilon c} = \frac{(1-\epsilon)b}{b} + \frac{1-\epsilon}{\epsilon} = \frac{1}{\epsilon} - \epsilon < \frac{2}{\epsilon} - 1$.

With any storage constraint in range $\big[a + (1-\epsilon)b + c, a + b + c\big)$, LMG will choose (1) which gives a total retrieval cost of $(1-\epsilon)c$. Note that with $\mathcal{S} < a + b + c$, LMG is not able to conduct step (2) after taking step (1). However, by choosing (2), which is also feasible, the total retrieval cost is $(1-\epsilon)b$. The proof is finished by observing c/b can be arbitrarily large. □

3.2 Hardness Results on General Graphs

The following theorem concerns MSR, where the objective is on the sum of retrieval cost. This theorem is derived from a natural reduction from Asymmetric k-Median.

Theorem 3.2. *On graphs with n nodes, even assuming single weight function and triangle inequality, there is no (α, β)-approximation for MinSum Retrieval if $\beta \le \frac{1}{2}(1-\epsilon)\Big(\ln n - \ln \alpha - O(1)\Big)$; in particular, for some constant c, there is no $(c \cdot n)$-approximation without relaxing storage constraint by some $\Omega(\log n)$ factor, unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$;*

For BMR, where the objective is on the storage cost and the constraints are on the maximum retrieval cost, the following hardness results follow from a reduction from Set Cover:

Theorem 3.3. *On both version graphs with n nodes, even assuming single weight function and triangle inequality, there is no $(c \ln n)$-approximation for BoundedMax Retrieval for any $c < 1$, unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$.*

Remark. While the hardness for our problems come from k-Median and Set Cover, it is not the case that we can leverage heuristics available for those problems since the reductions only transform an instance of k-Median (resp., Set Cover) to a very specific instance of our problem. We also provide the hardness for MMR and BSR in our full version.

3.3 Hardness on Arborescence

We show that MSR is NP-hard on arborescence instances. This essentially shows that our FPTAS algorithm in Section 5.1 is the best we can do in polynomial time. Detailed proof can be found in our full version.

Theorem 3.4. *On arborescence inputs, MinSum Retrieval is NP-hard even when assume single weight function and triangle inequality.*

4 EXACT ALGORITHM FOR BMR ON BIDIRECTIONAL TREES

In this section, we focus on the problem BMR, namely, we are given constraint $\mathcal{R}$ on the maximal retrieval cost, and want to minimize the total storage cost. We will provide the pseudo code of our complete algorithm in our full version.

Let $T = (V, E)$ be a bidirectional tree instance, namely, a tree-shaped directed graph (abbreviated "tree" in the rest of the section). Here, if $e = (u,v) \in E$, then $e' = (v,u) \in E$, though $s_e(r_e)$ might differ from $s_{e'}(r_{e'})$. We can arbitrarily pick a vertex v_{root} as root, and orient the tree such that the root has no parent, while all other nodes have exactly one parent. This process is straightforward, so we will assume that the given tree is rooted for the rest of the section.

For some $v \in V$, let $T_{[v]}$ denote the subtree of T rooted at v. If v is retrieved from materialized u, we use p_v^u to denote the parent of v on the unique $u - v$ path to retrieve v. We write $p_v^v = v$. We now describe a dynamic programming (DP) algorithm DP-BMR that solves BMR exactly on T.

DP variables. For $u, v \in V$, we define DP$[v][u]$ to be the minimum storage cost of a *partial solution* on $T_{[v]}$ with respect to version u. The partial solution is defined as a solution with all descendants of v are retrieved from some materialized node in $T_{[v]}$, while v is retrieved from a materialized version u, *potentially outside of the subtree* $T_{[v]}$. See Figure 3 for an illustration.

Importantly, also note that when calculating the storage cost for $DP[v][u]$, if u is not a part of $T_{[v]}$, the incident edge (p_v^u, v) is involved in the calculation, while other edges in the $u - v$ path, or the cost to materialize u, are not involved in it.

Base case. We iterate from the leaves up. Let $R(u,v)$ denote the retrieval cost of the path from u to v. For a leaf v, we set $DP[v][v] = s_v$, and $DP[v][u] = s_{(p_v^u, v)}$ for all $u \ne v$ with $R(u,v) \le \mathcal{R}$. Here, p_v^u is just the unique parent of v in the tree structure.

All choices of u, v such that $R(u,v) > \mathcal{R}$ are infeasible, and we therefore set DP$[v][u] = \infty$ in these cases.

Recurrence. For convenience, we define a helper variable $OPT[v]$ to be the minimum storage cost on the subproblem $T_{[v]}$, such that *v is either materialized or retrieved from one of its descendants.*[9] In other words,

$$OPT[v] = \min\{DP[v][w] : w \in V(T_{[v]})\}$$

For recurrence on DP$[v][u]$ such that $R(v,u) \le \mathcal{R}$, there are three possible cases of the relationship between v and u (see Figure 3 for illustration). In each case, we outline what we add to DP$[v][u]$ below.

Case 1. If $u = v$, we materialize v, and each child w of v can be either materialized, retrieved from their materialized descendants, or retrieved from the materialized $u = v$ (u for the other two cases). Note that this is exactly $\min\{OPT[w], \text{DP}[w][u]\}$, and similar facts hold for the following two cases as well.

[9]Note that the case where v is retrieved from u outside of $T_{[v]}$ is not considered in this helper variable.

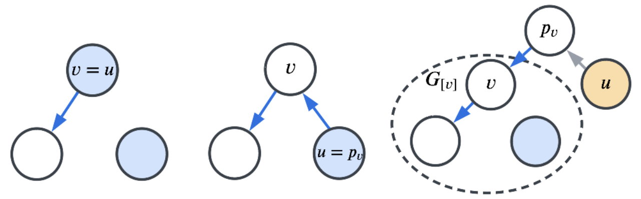

Figure 3: 3 cases of DP-BMR. The blue nodes and edges are stored in the partial solution.

Case 2. If $u \in V(T_{[v]})\backslash\{v\}$, we would store the edge (p_v^u, v). Note that p_v^u is a child of v and hence is also retrieved from the materialized u, so we must add $\text{DP}[p_v^u][u]$. We then add $\min\{OPT[w], \text{DP}[w][u]\}$ for all other children w of v.

Case 3. If $u \notin V(T_{[v]})$, we add the edge (p_v^u, v), where p_v^u is the parent of v in the tree structure. We then add $\min\{OPT[w], \text{DP}[w][u]\}$ for all children as before.

Output We output $\text{OPT}[v_{root}]$, which is the storage cost of the optimal solution. To output the configuration achieving this optimum, we can use the standard procedure where we store the configuration at each DP.

Theorem 4.1. *BoundedMax Retrieval is solvable on bidirectional tree instances in $O(n^2)$ time.*

Proof. **Time complexity**

It is straightforward to see that, for each v, u, computing $DP[v][u]$ takes $O(deg(v))$ time. Hence, computing all the dynamic programming table takes

$$\sum_{u\in V}\sum_{v\in V} deg(v) = \sum_{u\in V} O(n) = O(n^2).$$

Moreover, since the values $R(u, v)$ on a tree can be computed in $O(n^2)$ time, DP-BMR runs in $O(n^2)$ time.

Optimality We will show by induction that our DP table calculates optimal solution corresponding to each state, i.e., $DP[v][u]$ represents the total storage cost needed for $T_{[v]}$ given that v is retrieved from materialized u. Note that if $u \notin T_{[v]}$, then only the edge (p_v^u, v) is considered in $DP[v][u]$ among all edges in $u - v$ path in T.

In the base case on each leaf v, we set $DP[v][v] = s_v$, and $DP[v][u] = s_{(p_v^u, v)}$ if $u - v$ path has length at most $\mathcal{R}$. This is consistent with the optimal storage cost on the trivial subproblems.

Inductively, suppose we want to compute $DP[v][u]$. Notice that the storage needed for two children w, w' are independent from each other, implying that we can consider them separately. For each child w, if $u \notin T_{[w]}$, then w can either be retrieved through u or some other node in $T_{[w]}$. Hence, we add to $DP[v][u]$ the minimum between $OPT[w] = \min_{u' \in T_{[w]}} DP[w][u']$ and $DP[w][u]$. Otherwise, if $u \in T_{[w]}$, in order for v to be retrieved through u, w has to be retrieved through u, so we add $DP[w][u]$ to $DP[v][u]$. Finally, we increase $\text{DP}[v][u]$ by $s_{(p_v^u, v)}$ if $u \neq v$ and s_v if $u = v$. These are all possible cases. Given that $DP[w][u]$ is computed correctly, for all $w \in T_{[v]}$ and $u \in V$, then we compute $DP[v][u]$ correctly.

By induction, we conclude that the DP table is computed correctly. Since the table can capture all feasible solutions, it must capture an optimal solution as well. Hence, our algorithm outputs an optimal answer. □

5 FULLY POLYNOMIAL TIME APPROXIMATION SCHEME FOR MSR VIA DYNAMIC PROGRAMMING

In this section we focus on problem MSR and present a fully polynomial time approximation scheme (FPTAS) on digraphs whose *underlying undirected graph* has bounded treewidth. Similar techniques can be extended to all three other versions of the problems. However, we focus on the method itself and omit the details for the other three problems due to space constraints.

We start by describing a dynamic programming (DP) algorithm on trees in Section 5.1. In Section 5.2, we define all notations necessary for the latter subsection. Finally, in Section 5.3, we show how to extend our DP to the bounded treewidth graphs.

5.1 Warm-up: Bidirectional Trees

As shown in Theorem 3.1, the previously best-working LMG algorithm performs arbitrarily badly even on directed paths. In this section, as a warm-up to the more general algorithm in Section 5.3, we will present an FPTAS for bidirectional tree instances of MSR via dynamic programming. This algorithm also inspired a practical heuristic DP-MSR, presented in section Section 6.2.3.

We can WLOG assume the tree has a designated root v_{root} and a parent-child hierarchy. We can further assume that the tree is binary, via the standard trick of vertex splitting and adding edges of zero weight if necessary.

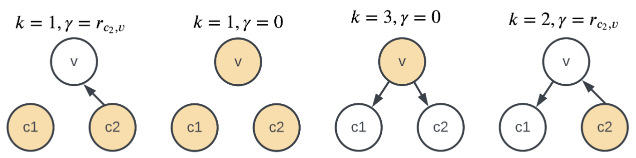

Figure 4: An illustration of DP variables in Section 5.1

DP variables. We explain the DP variables defined for MinSum Retrieval here: we define $\text{DP}[v][k][\gamma][\rho]$ to be the minimum storage cost for the subproblem with constraints v, k, γ, ρ such that (with examples illustrated in Figure 4)

(1) *Root for subproblem* $v \in V$ is a version on the tree; in each iteration, we consider the subtree rooted at v.

(2) *Dependency number* $k \in \mathbb{N}$ stands for the number of versions that will be retrieved via v (including v itself) in the subproblem solution. This is useful when calculating the extra retrieval cost incurred by retrieving v from its parent.

(3) *Root retrieval* $\gamma \in \mathbb{N}$ represents the cost of retrieving version v, the root in the subproblem. This is useful when calculating the extra retrieval cost incurred by retrieving the parent of v from v. Note that the root retrieval cost will be discretized, as specified later.

(4) *Total retrieval* $\rho \in \mathbb{N}$ represents the total retrieval cost of the subsolution. Similar to γ, ρ will also be discretized.

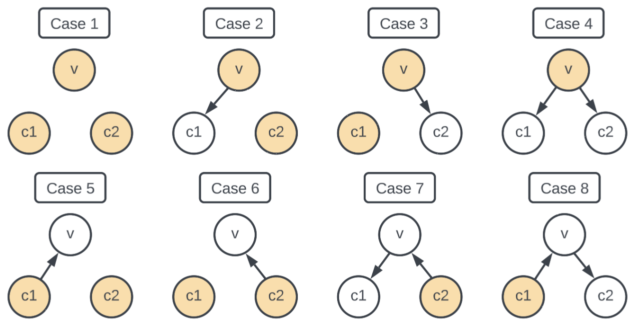

Figure 5: Eight types of connections on a binary tree. A node is colored if it is materialized or retrieved via delta from outside the chart. Otherwise, an uncolored node is retrieved from another node as illustrated with the arrows.

Discretizing retrieval costs. Let $r_{max} = \max_{e \in E}\{r_e\}$. The possible total retrieval cost ρ is within range $\{0, 1, \ldots, n^2 r_{max}\}$. To make the DP tractable, we partition this range further and define *approximated retrieval cost* $r'_{u,v}$ for edge $(u, v) \in E$ as follows:

$$r'_{u,v} = \lceil \frac{r_{u,v}}{l} \rceil \quad \text{where } l = \frac{n^2 r_{max}}{T(\epsilon)},\ T(\epsilon) = \frac{n^4}{\epsilon},$$

and $T(\epsilon)$ is the number of "ticks" we want to partition the retrieval range into. We will work with r' in the rest of the subsection.

Base case For a leaf v, we let $DP[v][1][0][0] = s_v$.

Recurrence step For each iteration, we take the minimum over all possible situations as illustrated in Figure 5. The recurrence relation for all cases will be given in our full version. Due to space limit, we only explain in detail the representative cases below:

5.1.1 ***Dealing with dependency.*** This refers to the case where a child is to be retrieved from its parent v. Consider case 2 in Figure 5 as an example. Note that $\gamma = 0$ in case 2 since v is materialized. The minimum storage cost in case 2 (given $v, k, \gamma = 0, \rho$) is:

$$\begin{aligned} S_2 = s_v + s_{v,c_1} - s_{c_1} \\ + \min_{\rho_1 \le \rho} \Bigg\{ DP[c_1][k-1][0][\rho_1 - (k-1) r'_{v,c_1}] \quad (1) \\ + \min_{k', \gamma_2}\{DP[c_2][k'][\gamma_2][\rho - \rho_1]\} \Bigg\} \quad (2) \end{aligned}$$

In Eq. (1), the dependency number for c_1 needs to be $k - 1$ for that of v to be k. The choice of ρ_1 determines how we are allocating retrieval costs budget ρ to c_1 and c_2 respectively. Specifically, the total retrieval cost allocated to subproblem on $G_{[c_1]}$ is $\rho_1 - (k - 1) \cdot r'_{v,c_1}$ since an extra $(k - 1) \cdot r'_{v,c_1}$ cost is incurred by the edge (v, c_1), as it is used $(k - 1)$ times by all versions depending on c_1.

In Eq. (2), for a given choice of ρ_1, we want to find the minimum storage cost over the dependency and retrieval cost of c_2. Minimizing this sum of Eqs. (1) and (2) over all possible combinations of budget splitting yields the minimum storage cost for case 2.

Uprooting We introduce *Uprooting*, a process extensively used in the next section. In the above example, the subproblem solution on $G_{[c_1]}$ materializes c_1, yet in case 2 we would replace this materialization with the diff (v, c_1). This explains the $-s_{c_1}$ term in the equation for S_2.

In general, the restriction of a global solution on a subproblem $G_{[v]}$ does not result in a feasible *partial solution*, due to the possibility of some $v \in V(G_{[z]})$ that's retrieved from versions outside of $G_{[z]}$. The uprooting process allows us to utilize the DP variables on the subproblem in this case. Conversely, by reversing this process (*un-uproot*), we can check if a subproblem is *compatible* with a bigger poblem. We explain this in more detail on Section 5.3.

Distributing dependency An additional complication arise in case 4, where v is required to have dependency number k and root retrieval 0. For each $k_1 + k_2 = k - 1$, we must go through subproblems where c_1 have dependency number k_1 and c_2 has that of k_2.

5.1.2 ***Dealing with retrieval.*** In contrast with dependencies, this refers to the case where v is retrieved from one of its children. We take case 5 as an example: given $v, k = 0, \gamma, \rho$,

$$\begin{aligned} S_5 = s_{c_1,v} + \min_{\rho_1 \le \rho} \Bigg\{ \min_{k_1}\{DP[c_1][k_1][\gamma - r_{c_1,v}][\rho_1 - \gamma]\} \\ + \min_{k_2,\gamma'}\{DP[c_2][k_2][\gamma'][\rho - \rho_1]\} \Bigg\} \end{aligned}$$

We allocate the retrieval cost similar to case 2. We will care less about the dependency number, over which we will take minimum. The retrieval cost for c_1 now has to be $\gamma - r_{c_1,v}$ since v has to be retrieved from c_1. Note importantly that now we are counting the retrieval cost for v in ρ_1, and so the retrieval cost remaining for the left subproblem now is $\rho_1 - \gamma$. Notice that since only one way of retrieving v will be stored, this retrieval cost will not be over-counted in any cases.

Similarly, we take minimum on all other unused parameters to get the best storage for case 5.

5.1.3 ***Combining the ideas.*** We take case 8 as an example where both retrieval and dependencies are involved. In case 8, v is retrieved from child c_1 (retrieval), and child c_2 is retrieved from v (dependency). Given v, k, γ, ρ, we claim that:

$$\begin{aligned} S_8 = s_{c_1,v} + s_{v,c_2} - s_{c_2} + \min_{\rho_1+\rho_2=\rho} \Bigg\{ \min_{k'}\{DP[c_1][k'][\gamma - r_{c_1,v}][\rho_1 - \gamma]\} \\ + DP[c_2][k-1][0][\rho_2 - (k-1)\cdot(r_2 + \gamma)] \Bigg\} \end{aligned}$$

Note that the c_1 side is identical to that for case 5. In combining both dependency and retrieval cases, there is slight adjustment in the dependency side: since v now might also depend on nodes further down c_1 side, the total extra retrieval cost created by adding edge (v, c_2) becomes $(k - 1) \cdot (r_2 + \gamma)$ instead of $(k - 1) \cdot (r_2)$.

Output Finally, with storage constraint $\mathcal{S}$ and root of the tree v_{root}, we output the configuration that outputs the minimum ρ among all cells satisfying $DP[v_{root}][\cdot][\cdot][\rho] \le \mathcal{S}$.

We shall formally state the following theorem.

Theorem 5.1. *For all $\epsilon > 0$, there is a $(1+\epsilon)$-approximation algorithm for MinSum Retrieval on bidirectional trees that runs in $O(n^{11}\epsilon^{-2})$.*

5.2 Treewidth-Related Definitions

We now consider a more general class of version graphs: any G whose *underlying undirected graph* G_0 has treewidth bounded by some constant k.

Definition 5.2 (Tree Decomposition [9]). A tree decomposition of an undirected graph $G_0 = (V_0, E_0)$ is a tree $T = (V_T, E_T)$, where each $z \in V_T$ is associated with a subset ("bag") S_z of V_0. The bags must satisfy the following conditions: (1) $\bigcup_{z \in V_T} S_z = V_0$, (2) For each $v \in V_0$, bags containing v induce a connected subtree of T, and (3) For each $e = (u, v) \in E_0$, there is a bag $z \in V_T$ such that $u, v \in S_z$.

The *width* of $T = (V_T, E_T)$ is $\max_{z \in V_T} |S_z| - 1$. The *treewidth* of undirected graph G_0 is the minimum width over all tree decompositions of G_0.

It follows that undirected forests have treewidth 1. We further note that there is also a notion of directed treewidth [39], but it's not suitable for our purpose.

For our purpose, we will WLOG assume a special kind of tree decomposition:

Definition 5.3 (Nice Tree Decomposition [13]). A nice tree decomposition is a tree decomposition with a designated root, where each node is one of the following types: (1) a **leaf**, which has no children, (2) a **separator**, which has one child whose bag is a subset of the child's bag, and (3) a **join**, which has two children, whose bag is exactly the union of its children's bags.

Given a bound k on the treewidth, there are multiple algorithms for calculating a desired tree decomposition of width k [7, 12, 27], or an approximation of k [8, 24, 26, 43]. For our case, the algorithm by Bodlaender [7] can be used to compute a tree decomposition in time $2^{O(k^3)} \cdot O(n)$, which is polynomial in n if the treewidth k of the given input is constant. Given such a tree decomposition, we can in $O(|V_0|)$ time find a nice tree decomposition of the same width with $O(k|V_0|)$ nodes [13].

5.3 Generalized Dynamic Programming

Here we outline the DP for MSR on graphs whose underlying undirected graph G_0 has treewidth at most $k-1$. In order to extend our algorithm in Section 5.1 to bounded treewidth graphs, we utilize the techniques from Hajiaghayi [34].

5.3.1 ***DP States.*** Similar to the warm-up, we will do the DP bottom-up on each $z \in V_T$ in the nice tree decomposition T. When we are at node z, let $V_{[z]} = \bigcup_{z' \in V(T_{[z]})} S_{z'}$ denote the set of vertices that were already considered bags up to bag S_z, including S_z. We now define the *DP states*. At a high level, each state describes some number of *partial solutions* on the subgraph $V_{[z]}$. When building a complete solution on G from the partial solutions, the state variables should give us *all* the information we need.

Each DP state on $z \in V_T$ consists of a tuple of functions

$$\mathcal{T}_z = (\mathrm{Par}_z, \mathrm{Dep}_z, \mathrm{Ret}_z, \mathrm{Anc}_z)$$

and a natural number ρ_z:

(1) *Parent function* $\mathrm{Par}_z : S_z \mapsto V_{[z]}$ describing the partial solution restricted on S_z. If $\mathrm{Par}_z(v) \neq v$ then v will be retrieved through the edge $(\mathrm{Par}_z(v), v)$. If $\mathrm{Par}_z(v) = v$ then v will be materialized.

(2) *Dependency function* $\mathrm{Dep}_z : S_z \mapsto [n]$. Similar to the dependency parameter in the warm-up, $\mathrm{Dep}_z(v)$ counts the number of nodes whose retrieval requires the retrieval of v.

(3) *Retrieval cost function* $\mathrm{Ret}_z : S_z \mapsto \{0, \ldots, K\}$. Similar to the root retrieval parameter in the warm-up, $\mathrm{Ret}_z(v)$ denotes the retrieval cost of version v in the partial solution on $V_{[z]}$.

(4) *Ancestor function* $\mathrm{Anc}_z : S_z \mapsto 2^{S_z}$. $u \in \mathrm{Anc}_z(v)$ denotes that u is retrieved in order to retrieve v. i.e. v is dependent on u. Different from the tree case, to produce a spanning forest here, we need this extra information to avoid directed cycles.

(5) ρ_z, the total retrieval cost of the subproblem according to the partial solution. Similar to its counterpart in the warm-up, ρ_z will be discretized by the same technique that makes the approximation an FPTAS.

A feasible state on $z \in V_T$ is a pair $(\mathcal{T}_z, \rho_z)$ as defined, which correctly describes some partial solution on $V_{[z]}$ whose retrieval cost is exactly ρ_z. Each state is further associated with a storage value $\sigma(\mathcal{T}_z, \rho_z) \in \mathbb{Z}^+$, indicating the minimum storage needed to achieve the state $(\mathcal{T}_z, \rho_z)$ on $V_{[z]}$. We call a minimum-storage solution "the partial solution $\mathcal{T}_z$" for convenience.

Since we have described our states, we are now ready to describe how to compute each state.

5.3.2 ***Recurrence on leaves.*** For each leaf $z \in V_T$, we can enumerate all possible choices of Par_z on S_z. It's easy to calculate the corresponding Dep_z, Ret_z, and Anc_z functions, as well as the retrieval cost ρ_z and storage cost $\sigma(\mathcal{T}_z, \rho_z)$ for each choice of Par_z. These are all the feasible states.

5.3.3 ***Recurrence on separators.*** On a separator z with child c, because $S_z \subseteq S_c$, the feasible states on z are just restrictions of those on c:

$$\sigma(\mathcal{T}_z, \rho_z) = \min\{\sigma(\mathcal{T}_c, \rho_z) : \mathcal{T}_c\big|_{S_z} = \mathcal{T}_z\}$$

where $\mathcal{T}_c\big|_{S_z}$ is the natural restriction of $\mathcal{T}_c$ on S_z:

$$\mathcal{T}_c\big|_{S_z} = \left(\mathrm{Par}_c\big|_{S_z}, \mathrm{Dep}_c\big|_{S_z}, \mathrm{Ret}_c\big|_{S_z}, \mathrm{Anc}_c\big|_{S_z}\right)$$

where $\mathrm{Anc}_c\big|_{S_z}$ is Anc_c restricted on S_z with the additional requirement that its output is also an intersection with S_z. (So that the range of Anc_z is 2^{S_z}, as in the definition.)

5.3.4 ***Recurrence on joins.*** Suppose we are at a join z with children a, b, where $S_z = S_a \cup S_b$.

Compatibility. Naturally, we want to find all partial solutions $(\mathcal{T}_a, \rho_a)$ and $(\mathcal{T}_b, \rho_b)$ that combine into a given $(\mathcal{T}_z, \rho_z)$, and then take the minimum over the objective of all combinations. The first attempt is to consider the states $\mathcal{T}_a, \mathcal{T}_b$ to be $\mathcal{T}_z\big|_{S_a}$ and $\mathcal{T}_z\big|_{S_b}$. However, in $\mathcal{T}_z$ there could be $v \in S_a$ such that $\mathrm{Par}_z(v) \in V_{[b]} \setminus S_a$, as in node u of Figure 6. To resolve this, we apply similar ideas to the *uprooting* process in Section 5.1, and pick $\mathcal{T}_a$ which materializes this v and fits the restriction of the solution $\mathcal{T}_z$ on the subproblem $V_{[a]}$, as in (b) of Figure 6. We say a state tuple $(\mathcal{T}_a, \mathcal{T}_b)$ is *compatible* with

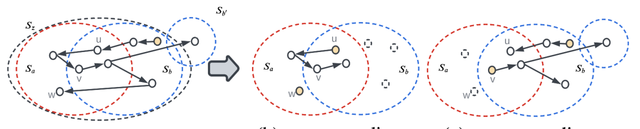
(a). original path on z
(b). corresponding partial solution on a
(c). corresponding partial solution on b

Figure 6: Illustration for compatibility. A node is colored if it is materialized.

$\mathcal{T}_z$ if they can combine into $\mathcal{T}_z$. Let COMPATIBILITY[10] be a precedure that takes $\mathcal{T}_a, \mathcal{T}_b, \mathcal{T}_z$ and returns `true` iff $(\mathcal{T}_a, \mathcal{T}_b)$ is compatible with $\mathcal{T}_z$.

Checking compatibility and the Un-Uprooting process. COMPATIBILITY checks whether $\mathcal{T}_a, \mathcal{T}_b$ are indeed the 'restrictions" of $\mathcal{T}_z$ on S_a and S_b in Figure 6's sense. To give a high-level idea, when calculating the restriction of $\mathcal{T}_z$ on $\mathcal{T}_a$, four edge cases are considered and analyzed separately, as illustrated in Figure 7. Here we briefly outline the steps.

From the above definition of compatibility, we note that for each $v \in S_a$, we have either $\text{Par}_z(v) = \text{Par}_a(v)$, or $\text{Par}_z(v) \in V_{[b]} \setminus S_a$ is a node from b's side. In the latter case, we call such v an "uprooted node", and we denote the set of all uprooted nodes in S_a (resp. S_b) as U_a (resp. U_b). As an example, U_a comprises of vertices u, w and U_b has vertex v in the case of Figure 6. This procedure of finding such U_a, U_b is defined SCAN UPROOTED NODES[10]. After obtaining U_a, U_b, the subroutine UN-UPROOT[10] is called. It reverses the idea of "uprooting" as described in the warm-up. We refer to the full version for details.

The next step in COMPATIBILITY is to distribute the external dependencies to $V_{[a]} \setminus S_a$ and $V_{[b]} \setminus S_b$, much like how we distribute dependency number k to the two children in case 4 of Figure 5.

Calculating ρ and recurrence Given that $(\mathcal{T}_a, \mathcal{T}_b)$ are compatible with $\mathcal{T}_z$, we want to find the objective, $\sigma(\mathcal{T}_z, \rho_z)$, with the recurrence relation involving $\sigma(\mathcal{T}_a, \rho_a) + \sigma(\mathcal{T}_b, \rho_b)$ for suitable ρ_a and ρ_b. However, we can't simply take $\rho_a + \rho_b = \rho_z$ due to the combining processes above. We thus implement DISTRIBUTE RETRIEVAL[10] to calculate ρ_Δ such that $\rho_a + \rho_b = \rho_z - \rho_\Delta$ and then iterate through all satisfying ρ_a and ρ_b.

Recurrence relation. Finally, we have all we need for the recurrence relation:

$$\sigma(\mathcal{T}_z, \rho_z) = \min\{\sigma(\mathcal{T}_a, \rho_a) + \sigma(\mathcal{T}_b, \rho_b) - \mathit{uproot} - \mathit{overcount}\}$$

where the minimum is taken over all $(\mathcal{T}_a, \mathcal{T}_b)$ that are compatible with $\mathcal{T}_z$ and all $\rho_a + \rho_b = \rho_z - \rho_\Delta$, and

$$\mathit{uproot} = \sum_{v \in U_a} (s_v - s_{\text{Par}_z(v),v}) + \sum_{v \in U_b} (s_v - s_{\text{Par}_z(v),v}), \text{ and}$$

$$\mathit{overcount} = \sum_{v \in S_a \cap S_b} s_{\text{Par}_z(v),v}.$$

If k is constant, then both recurrence relations take $\text{poly}(n)$ time. This is because there are $\text{poly}(n)$ many states on a and b, and it takes $\text{poly}(n)$ steps to check the compatibility of $(\mathcal{T}_a, \mathcal{T}_b)$ with $\mathcal{T}_z$ and compute ρ_Δ.

[10]We will provide pseudocode in the full version.

Output The minimum storage cost of a global solution is hence just $\min\{\sigma(\mathcal{T}_z), \rho_z\}$ over all states $\mathcal{T}_z$ and ρ_z, where z is the designated root of the nice tree decomposition.

We conclude this section with the following theorem.

THEOREM 5.4. *For a constant $k \geq 1$, on the set of graphs whose undelying undirected graph has treewidth at most k, MINSUM RETRIEVAL admits an FPTAS, while BOUNDEDSUM RETRIEVAL has an $(1, 1+\epsilon)$ bi-criteria approximation that finishes in $\text{poly}(n, \frac{1}{\epsilon})$ time.*

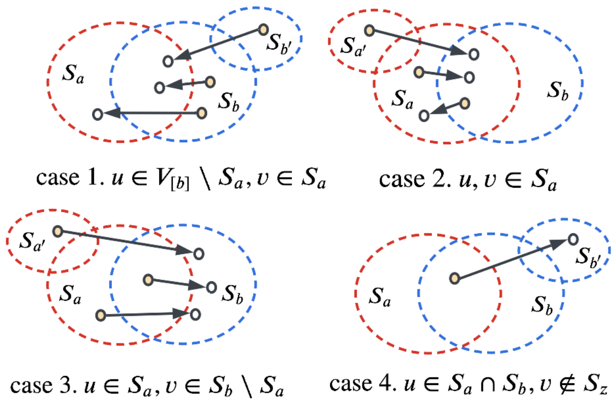
case 1. $u \in V_{[b]} \setminus S_a, v \in S_a$
case 2. $u, v \in S_a$
case 3. $u \in S_a, v \in S_b \setminus S_a$
case 4. $u \in S_a \cap S_b, v \notin S_z$

Figure 7: Four types of edge (u, v) involved when restricting $\mathcal{T}_z$ to $\mathcal{T}_a$.

6 EXPERIMENTS AND IMPROVED HEURISTICS FOR MSR AND BMR

In this section, we propose three new heuristics that are inspired by empirical observations and theoretical results. We will discuss the experimental setup, datasets used, and experimental results for empirical validation of the performance of the algorithms. The performance and run time of these new algorithms are compared with previous best-performing heuristics[11].

In all figures, the vertical axis (objective and run time) is presented in *logarithmic scale*. Run time is measured in *milliseconds*.

6.1 Datasets and Construction of Graphs

We use real-world GitHub repositories of varying sizes as datasets, from which we construct version graphs. Each commit corresponds to a node with its weight (storage cost) equal to its size in bytes. Between each pair of parent and child commits, we construct bidirectional edges. The storage and retrieval costs of the edges are calculated, in bytes, based on the actions (such as addition, deletion, and modification of files) required to change one version to the other in the direction of the edge. We use simple `diff` to calculate the deltas, hence the storage and retrieval costs are proportional to each other. Graphs generated this way are called "**natural graphs**" in the rest of the section.

In addition, we also aim to test (1) the cases where the retrieval and storage costs of an edge can greatly differ from each other, and (2) the effect of tree-like shapes of graphs on the performance of

[11]Our code can be found at https://github.com/Sooooffia/Graph-Versioning.

algorithms. Therefore, we also conduct experiments on modified graphs in the following two ways:

(1) **Random compression.** We simulate compression of data by scaling storage cost with a random factor between 0.3 and 1, and increasing the retrieval cost by 20% (for de-compression). In realistic cases, storage and retrieval costs for the "delta" between two versions are often proportional, but randomness is added for generality of our experiments.

(2) **ER construction.** Instead of the naturally constructing edges between each pair of parent-child commits, we construct the edges as in an Erdős-Rényi random graph: between each pair (u, v) of nodes, with probability p both deltas (u, v) and (v, u) are constructed, and with probability $1-p$ neither are constructed. This creates graphs much less tree-like than the natural construction. In particular, ER graphs have treewidth $\Theta(n)$ with high probability if the number of edges per vertex is greater than a small constant [29].

The datasets we use are from GitHub's repositories, namely, `datasharing`[12], `LeetCodeAnimation`[13], `styleguide`[14], `996.ICU`[15], and `freeCodeCamp`[16]. The characteristic of these datasets can be found in Table 4.

Dataset	#nodes	#edges	avg. materialization	avg. storage
datasharing	29	74	7672	395
styleguide	493	1250	1.4e6	8659
996.ICU	3189	9210	1.5e7	337038
freeCodeCamp	31270	71534	2.5e7	14800
LeetCodeAnimation	246	628	1.7e8	1.2e7
LeetCode 0.05	246	3032	1.7e8	1.0e8
LeetCode 0.2	246	11932	1.7e8	1.0e8
LeetCode 1	246	60270	1.7e8	1.0e8

Table 4: Natural and ER graphs overview.

We ran the experiments on a PC with an Intel i9-13900K and 64GB RAM. We used the python packages Networkx [33] and GitPython to generate version graphs. All algorithms were implemented using C++. To compute minimum spanning arborescence, we applied Gabow et al.'s algorithm [28], and we used Böther et al.'s code for implementation [15].

6.2 Algorithms Implementation

6.2.1 ***Baselines.*** In considering MSR and BMR, we used LMG (refer to Section 3.1) and Modified Prim(MP) [11] as the respective baselines in assessing the performance of algorithms. Both are best-performing heuristics in previous experiments for the respective problems [11].

6.2.2 ***LMGA: improving LMG.*** For MSR, we implemented a modification for LMG named LMG-All (LMGA). Instead of searching for the most efficient version to materialize per step, we can enlarge the scope of this search to explore the payoff of modifying any single edge.

Specifically, we will keep a set of active edges E_{active}, initialized to $E(G_{aux})$. On each iteration, let $\text{Par}(v)$ be the parent of v in the current solution T. For all $e = (u, v) \in E_{active}$, we will consider the potential solution T_e, obtained by adding the edge e and removing $(\text{Par}(v), v)$ from T. We then update T to T_{e^*} and remove e^* from E_{active}, where e^* maximizes $\rho_e = \frac{R(T)-R(T_e)}{S(T_e)-S(T)}$.

While LMGA considers more edges than LMG, it is not obvious that LMGA always provides a better solution. Due to its greedy nature, the first move might be better, but it may possibly be stuck in a worse local optimum.

6.2.3 ***DP heuristics.*** We also propose DP heuristics on both MSR and BMR, as inspired by algorithms in Sections 4 and 5. Importantly, we note that DP algorithms proposed for bounded treewidth graphs involve the calculation of nice tree decompositions and complicated subroutines (UN-UPROOTING, etc.), which severely slow down the running time on graphs with high treewidths. To speed up the algorithm, we instead only run the DP on bi-directional trees (namely, with treewidth 1) extracted from our general input graphs, with the steps below:

(1) Calculate a minimum spanning arborescence A of the graph G rooted at the first commit v_1. We use the sum of retrieval and storage costs as weight.

(2) Generate a bidirectional tree G' from A. Namely, we have $(u, v), (v, u) \in E(G')$ for each edge $(u, v) \in E(A)$.

(3) Run the proposed DP for MSR and BMR on directed trees (see Section 5.1 and Section 4) with input G', and return the solution.

In addition, we also implement the following modifications for *MSR* to further speed up the algorithm:

(1) Total *storage* cost is discretized instead of retrieval cost, since the former generally has a smaller range.

(2) Geometric discretization is used instead of linear discretization.

(3) A pruning step is added, where the DP variable discards all subproblem solutions whose storage cost exceeds some bound.

All three original features are necessary in the proof for our theoretical results, but in practice, the modified implementations show comparable results but significantly improves the running time.

6.3 Results and Discussion

6.3.1 ***Results in MSR.*** Section 6.3.1, Figure 9, and Figure 10 demonstrate the performance of the three MSR algorithms on natural graphs, randomly compressed natural graphs, and random compression ER graphs. The running times for the algorithms are shown in Figure 9 and Figure 10. Note since run time for most non-ER graphs exhibit similar trends, many are omitted here due to space constraint. Also note that, since all data points by the DP-MSR are generated with a single run of the DP, its running time is shown as a horizontal line over the full range for storage constraint.

We run DP-MSR with $\epsilon = 0.05$ on most graphs, except $\epsilon = 0.1$ for `freeCodeCamp` (for the feasibility of run time). The pruning value for DP variables is at twice the minimum storage for uncompressed graphs, and ten times the minimum storage for randomly compressed graphs.

Performance analysis. On most graphs, DP-MSR outperforms LMGA, which in turn outperforms LMG. This is especially clear on natural version graphs, where DP-MSR solutions are near 1000

[12]https://github.com/jtleek/datasharing
[13]https://github.com/MisterBooo/LeetCodeAnimation
[14]https://github.com/google/styleguide
[15]https://github.com/996icu/996.ICU
[16]https://github.com/freeCodeCamp/freeCodeCamp

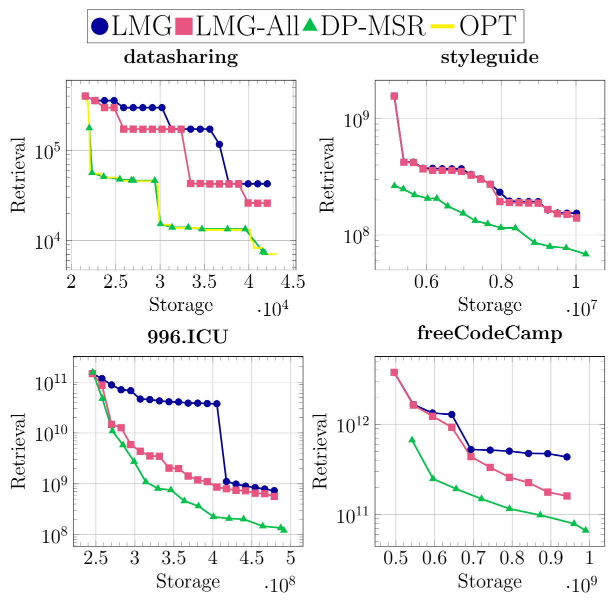

Figure 8: Performance of MSR algorithms on natural graphs. OPT is obtained by solving an integer linear program (ILP) using Gurobi [31]. ILP takes too long to finish on all graphs except datasharing.

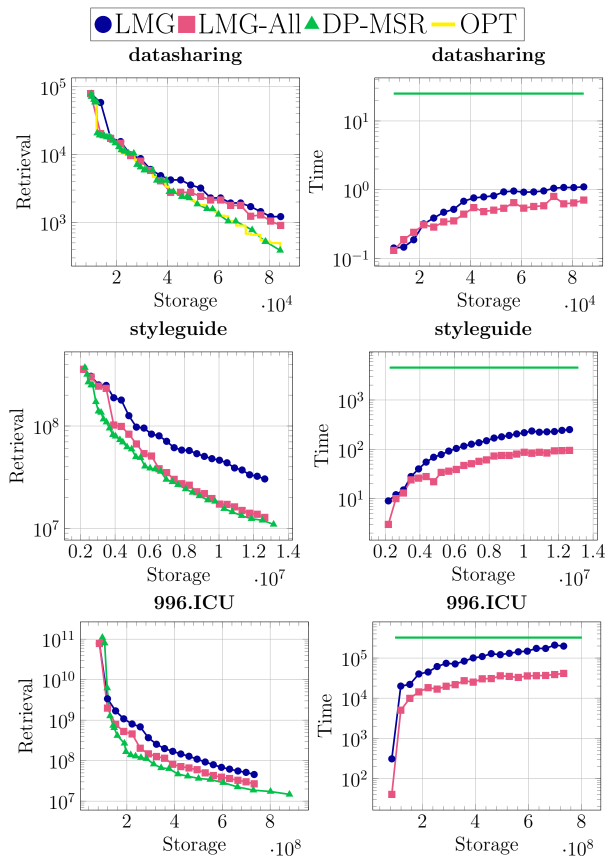

Figure 9: Performance and run time of MSR algorithms on compressed graphs.

times better than LMG solutions on 996.ICU. in Section 6.3.1. On datasharing, DP-MSR almost perfectly matches the optimal solution for all constraint ranges.

On naturally constructed graphs (Section 6.3.1), LMGA often has comparable performance with LMG when storage constraint is low. This is possibly because both algorithms can only iterate a few times when the storage constraint is almost tight. DP-MSR, on the other hand, performs much better on natural graphs even for low storage constraint.

On graphs with simulated random compression (Figure 9), the dominance of DP in performance over the other two algorithms become less significant. This is anticipated because of the fact that DP only runs on a subgraph of the input graph. Intuitively, most of the information is already contained in a minimum spanning tree when storage and retrieval costs are proportional. Otherwise, the dropped edges may be useful. (They could have large retrieval but small storage, and vice versa.)

Finally, LMG's performance relative to our new algorithms is much worse on ER graphs. This may be due to the fact that LMG cannot look at non-auxiliary edges once the minimum arborescence is initialized, and hence losing most of the information brought by the extra edges. (Figure 10).

Run time analysis. For all natural graphs, we observe that LMGA uses no more time than LMG (as shown in Figure 9). Moreover, LMGA is significantly quicker than LMG on large natural graphs, which was unexpected considering that the two algorithms have almost identical structures in implementation. Possibly, this can be due to the fact that LMG makes bigger, more expensive changes on each iteration (materializing a node with many dependencies, for instance) as compared to LMGA.

As expected, though, LMGA takes much more time than the other two algorithms on denser ER graphs (Figure 10), due to the large number of edges.

DP-MSR is often slower than LMG, except when ran on the natural construction of large graphs (Figure 9). However, unlike LMG and LMGA, the DP algorithm returns a whole spectrum of solutions at once, so it's difficult to make a direct comparison between the two. We also note that the runtime of DP heavily depends on the choice of ϵ and the storage pruning value. Hence, the user can trade-off the runtime with solution's qualities by parameterize the algorithm with coarser configuration (i.e., higher ϵ).

6.3.2 *Results in BMR.*

As compared to MSR algorithms, the performance and run time of our BMR algorithms are much more predictable and stable. They exhibit similar trends across different ways of graph construction as mentioned in earlier sections - including the non-tree-like ER graphs, surprisingly.

Due to space limitation, we present the results on natural graphs, as shown in Figure 11, to respectively illustrate their performance and run time.

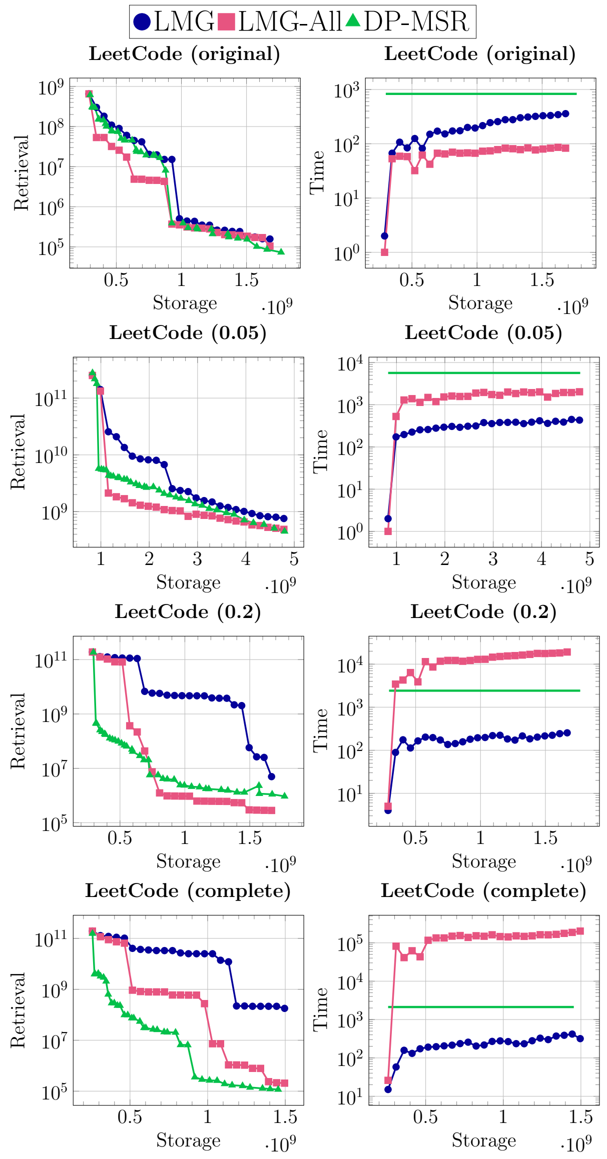

Figure 10: Performance and run time of MSR algorithms on compressed ER graphs.

Performance analysis. For every graph we tested, DP-BMR outperforms MP on most of the retrieval constraint ranges. As the retrieval constraint increases, the gap between MP and DP-BMR solution also increases. We also observe that DP-BMR performs worse than MP when the retrieval constraint is at zero. This is because the bidirectional tree have fewer edges than the original graph. (Recall that the same behavior happened for DP-MSR on compressed graphs)

We also note that, unlike MP, the objective value of DP-BMR solution monotonically decreases with respect to retrieval constraint. This is again expected since they are essentially optimal solutions the problem on the bidirectional tree.

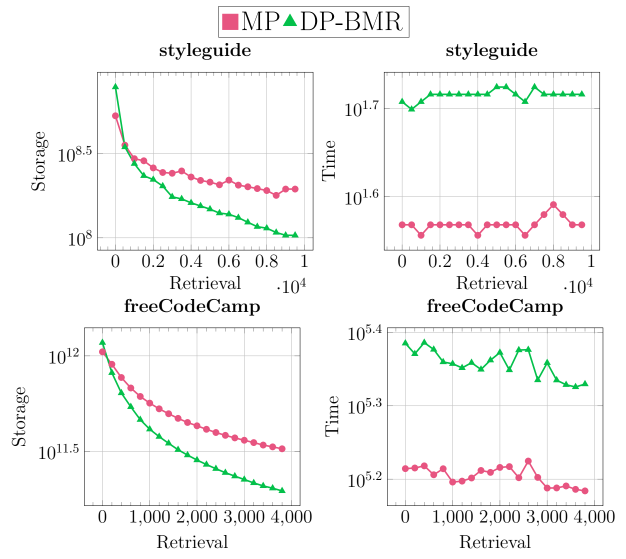

Figure 11: Performance and run time of BMR algorithms on natural version graphs.

Run time analysis. For all graphs, the runtimes of DP-BMR and MP are comparable within a constant factor. This is true with varying graph shapes and construction methods in all our experiments, and representative data is exhibited in Figure 11. Unlike LMG and LMGA, their runtimes do not change much with varying constraint values.

6.3.3 *Overall Evaluation.*

For MSR, we recommend always using one of LMGA and DP-MSR in place of LMG for practical use. On sparse graphs, LMGA dominates LMG both in performance and run time. DP-MSR can also provide a frontier of better solutions in a reasonable amount of time, regardless of the input.

For BMR, DP-BMR usually outperforms MP, except when the retrieval constraint is close to zero. Therefore, we recommend using DP in most situations.

7 CONCLUSION

In this paper, we developed fully polynomial time approximation algorithms for graphs with bounded treewidth . This often captures the typical manner in which edit operations are applied on versions. However, due to the high complexity of these algorithms, they are not yet practical to handle the size of real-life graphs. On the other hand, we extracted the idea behind this approach as well as previous LMG approach, and developed two heuristics which significantly improved both the performance and run time in experiments.

Future Works. There are many possible future directions. For one, a polynomial-time algorithm with bounded approximation ratio for general graph is desirable. Even for restricted classes of graphs, any development of a practical algorithm that can handle larger scale graphs is interesting. Moreover, in enterprise setting, often some versions are requested more often than others. It would be interesting to extend our work to handle such use cases.

REFERENCES

[1] 2005. Git. https://github.com/git/git. last accessed: 13-Oct-22.

[2] 2016. Pachyderm. https://github.com/pachyderm/pachyderm. last accessed: 13-Oct-22.

[3] 2017. DVC. https://github.com/iterative/dvc. last accessed: 13-Oct-22.

[4] 2019. Dolt. https://github.com/dolthub/dolt. last accessed: 13-Oct-22.

[5] 2019. TerminusDB. https://github.com/terminusdb/terminusdb. last accessed: 13-Oct-22.

[6] 2020. LakeFS. https://github.com/treeverse/lakeFS. last accessed: 13-Oct-22.

[7] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. 1987. Complexity of Finding Embeddings in a k-Tree. *SIAM Journal on Algebraic Discrete Methods* 8, 2 (1987), 277–284. https://doi.org/10.1137/0608024 arXiv:https://doi.org/10.1137/0608024

[8] Mahdi Belbasi and Martin Fürer. 2021. Finding All Leftmost Separators of Size $\leq k$. In *Combinatorial Optimization and Applications: 15th International Conference, COCOA 2021, Tianjin, China, December 17–19, 2021, Proceedings* (Tianjin, China). Springer-Verlag, Berlin, Heidelberg, 273–287. https://doi.org/10.1007/978-3-030-92681-6_23

[9] Umberto Bertelè and Francesco Brioschi. 1973. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A* 14, 2 (1973), 137–148. https://doi.org/10.1016/0097-3165(73)90016-2

[10] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. 2014. DataHub: Collaborative Data Science & Dataset Version Management at Scale. *arXiv* (2014). https://doi.org/10.48550/arxiv.1409.0798 arXiv:1409.0798

[11] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya G. Parameswaran. 2015. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *Proc. VLDB Endow.* 8, 12 (2015), 1346–1357. https://doi.org/10.14778/2824032.2824035

[12] Hans L. Bodlaender. 1993. A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '93)*. Association for Computing Machinery, New York, NY, USA, 226–234. https://doi.org/10.1145/167088.167161

[13] Hans L. Bodlaender. 1998. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science* 209, 1 (1998), 1–45. https://doi.org/10.1016/S0304-3975(97)00228-4

[14] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. *2020 IEEE 36th International Conference on Data Engineering (ICDE)* 00 (2020), 709–720. https://doi.org/10.1109/icde48307.2020.00067 arXiv:2011.10427

[15] Maximilian Böther, Otto Kißig, and Christopher Weyand. 2023. Efficiently Computing Directed Minimum Spanning Trees. In *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 86–95.

[16] Dan Brickley, Matthew Burgess, and Natasha Noy. 2019. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. *The World Wide Web Conference* (2019), 1365–1375. https://doi.org/10.1145/3308558.3313685

[17] Jackson Brown and Nicholas Weber. 2021. DSDB: An Open-Source System for Database Versioning & Curation. *2021 ACM/IEEE Joint Conference on Digital Libraries (JCDL)* 00 (2021), 299–307. https://doi.org/10.1109/jcdl52503.2021.00044

[18] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2000. Data Provenance: Some Basic Issues. *Lecture Notes in Computer Science* (2000), 87–93. https://doi.org/10.1007/3-540-44450-5_6

[19] Randal C. Burns and Darrell D. E. Long. 1998. In-place reconstruction of delta compressed files. *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing - PODC '98* (1998), 267–275. https://doi.org/10.1145/277697.277747

[20] Amit Chavan and Amol Deshpande. 2017. DEX: Query Execution in a Delta-based Storage System. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), 171–186. https://doi.org/10.1145/3035918.3064056

[21] Markus Chimani and Joachim Spoerhase. 2015. Network Design Problems with Bounded Distances via Shallow-Light Steiner Trees. In *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Ernst W. Mayr and Nicolas Ollinger (Eds.), Vol. 30. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 238–248. https://doi.org/10.4230/LIPIcs.STACS.2015.238

[22] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. 2022. Materialization and Reuse Optimizations for Production Data Science Pipelines *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1962–1976. https://doi.org/10.1145/3514221.3526186

[23] David Eppstein. 1992. Parallel recognition of series-parallel graphs. *Information and Computation* 98, 1 (1992), 41–55.

[24] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. 2005. Improved Approximation Algorithms for Minimum-Weight Vertex Separators. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing* (Baltimore, MD, USA) *(STOC '05)*. Association for Computing Machinery, New York, NY, USA, 563–572. https://doi.org/10.1145/1060590.1060674

[25] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Sam Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), 1001–1012. https://doi.org/10.1109/icde.2018.00094

[26] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, MichaŁ Pilipczuk, and Marcin Wrochna. 2018. Fully Polynomial-Time Parameterized Computations for Graphs and Matrices of Low Treewidth. *ACM Trans. Algorithms* 14, 3, Article 34 (jun 2018), 45 pages. https://doi.org/10.1145/3186898

[27] Fedor V. Fomin, Ioan Todinca, and Yngve Villanger. 2015. Large Induced Subgraphs via Triangulations and CMSO. *SIAM J. Comput.* 44, 1 (2015), 54–87. https://doi.org/10.1137/140964801 arXiv:https://doi.org/10.1137/140964801

[28] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* 6, 2 (1986), 109–122.

[29] Yong Gao. 2012. Treewidth of Erdős–Rényi random graphs, random intersection graphs, and scale-free random graphs. *Discrete Applied Mathematics* 160, 4-5 (2012), 566–578.

[30] Rohan Ghuge and Viswanath Nagarajan. 2022. Quasi-polynomial algorithms for submodular tree orienteering and directed network design problems. *Mathematics of Operations Research* 47, 2 (2022), 1612–1630.

[31] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[32] Bernhard Haeupler, D. Ellis Hershkowitz, and Goran Zuzic. 2021. Tree Embeddings for Hop-Constrained Network Design. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 356–369. https://doi.org/10.1145/3406325.3451053

[33] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[34] Mohammad Taghi Hajiaghayi. 2001. *Algorithms for graphs of (locally) bounded treewidth*. Ph.D. Dissertation. Citeseer.

[35] Mohammad Taghi Hajiaghayi, Guy Kortsarz, and Mohammad R Salavatipour. 2009. Approximating buy-at-bulk and shallow-light k-Steiner trees. *Algorithmica* 53, 1 (2009), 89–103.

[36] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. 2020. ORPHEUSDB: bolt-on versioning for relational databases (extended version). *The VLDB Journal* 29, 1 (2020), 509–538. https://doi.org/10.1007/s00778-019-00594-5

[37] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. 1998. Delta algorithms: an empirical analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 2 (1998), 192–214. https://doi.org/10.1145/279310.279321

[38] Yasith Jayawardana and Sampath Jayarathna. 2019. DFS: A Dataset File System for Data Discovering Users. *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)* 00 (2019), 355–356. https://doi.org/10.1109/jcdl.2019.00068 arXiv:1905.13363

[39] Thor Johnson, Neil Robertson, P. D. Seymour, and Robin Thomas. 2001. Directed tree-width. *Journal of Combinatorial Theory. Series B* 82, 1 (May 2001), 138–154. https://doi.org/10.1006/jctb.2000.2031 Funding Information: 1Partially supported by the NSF under Grant DMS-9701598. 2 Research partially supported by the DIMACS Center, Rutgers University, New Brunswick, NJ 08903. 3Partially supported by the NSF under Grant DMS-9401981. 4Partially supported by the ONR under Contact N00014-97-1-0512. 5Partially supported by the NSF under Grant DMS-9623031 and by the NSA under Contract MDA904-98-1-0517.

[40] M. Reza Khani and Mohammad R. Salavatipour. 2016. Improved Approximations for Buy-at-Bulk and Shallow-Light k-Steiner Trees and (k,2)-Subgraph. *J. Comb. Optim.* 31, 2 (feb 2016), 669–685. https://doi.org/10.1007/s10878-014-9774-5

[41] Samir Khuller, Balaji Raghavachari, and Neal E. Young. 1995. Balancing Minimum Spanning Trees and Shortest-Path Trees. *Algorithmica* 14, 4 (1995), 305–321. https://doi.org/10.1007/BF01294129

[42] Udayan Khurana and Amol Deshpande. 2012. Efficient Snapshot Retrieval over Historical Graph Data. *arXiv* (2012). https://doi.org/10.48550/arxiv.1207.5777 arXiv:1207.5777 Graph database systems — stroing dynamic graphs so that a graph at a specific time can be queried. Vertices are marked with bits encoding information on which versions it belong to.

[43] Tuukka Korhonen. 2022. A Single-Exponential Time 2-Approximation Algorithm for Treewidth. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. 184–192. https://doi.org/10.1109/FOCS52979.2021.00026

[44] Guy Kortsarz and David Peleg. 1997. Approximating shallow-light trees. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. 103–110.

[45] Josh MacDonald. 2000. *File system support for delta compression*. Ph.D. Dissertation. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkley.

[46] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The Relational Dataset Branching System. *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases* 9, 9 (2016), 624–635. https://doi.org/10.14778/2947618.2947619

[47] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. 2022. CHEX: multiversion replay with ordered checkpoints. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1297–1310. https://doi.org/10.14778/3514061.3514075

[48] Madhav V Marathe, Ramamoorthi Ravi, Ravi Sundaram, SS Ravi, Daniel J Rosenkrantz, and Harry B Hunt III. 1998. Bicriteria network design problems. *Journal of algorithms* 28, 1 (1998), 142–171.

[49] Koyel Mukherjee, Raunak Shah, Shiv K. Saini, Karanpreet Singh, Khushi , Harsh Kesarwani, Kavya Barnwal, and Ayush Chauhan. 2023. Towards Optimizing Storage Costs on the Cloud. *IEEE 39th International Conference on Data Engineering (ICDE) (To Appear)* (2023).

[50] William Nagel. 2006. Subversion: not just for code anymore. *Linux Journal* 2006, 143 (2006), 10.

[51] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (aug 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[52] R. Ravi. 1994. Rapid rumor ramification: approximating the minimum broadcast time. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 202–213. https://doi.org/10.1109/SFCS.1994.365693

[53] Paul Roome, Tao Feng, and Sachin Thakur. 2022. Announcing the Availability of Data Lineage With Unity Catalog. https://www.databricks.com/blog/2022/06/08/announcing-the-availability-of-data-lineage-with-unity-catalog.html. last accessed: 13-Oct-22.

[54] Maximilian E Schule, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. 2019. Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. *Proceedings of the 31st International Conference on Scientific and Statistical Database Management* (2019), 169–180. https://doi.org/10.1145/3335783.3335792

[55] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. 2012. Efficient Versioning for Scientific Array Databases. *2012 IEEE 28th International Conference on Data Engineering* 1 (2012), 1013–1024. https://doi.org/10.1109/icde.2012.102

[56] Yogesh L Simmhan, Beth Plale, Dennis Gannon, et al. [n.d.]. A survey of data provenance techniques. ([n. d.]).

[57] Dimitre Trendafilov Nasir Memon Torsten Suel. 2002. zdelta: An efficient delta compression tool. (2002).

[58] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. 2018. Forkbase: An Efficient Storage Engine for Blockchain and Forkable Applications. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1137–1150. https://doi.org/10.14778/3231751.3231762

[59] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. 2014. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation* 79 (2014), 258–272. https://doi.org/10.1016/j.peva.2014.07.016

[60] Tangwei Ying, Hanhua Chen, and Hai Jin. 2020. Pensieve: Skewness-Aware Version Switching for Efficient Graph Processing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 699–713. https://doi.org/10.1145/3318464.3380590

[61] Yin Zhang, Huiping Liu, Cheqing Jin, and Ye Guo. 2018. Storage and Recreation Trade-Off for Multi-version Data Management. In *Web and Big Data*, Yi Cai, Yoshiharu Ishikawa, and Jianliang Xu (Eds.). Springer International Publishing, Cham, 394–409.