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

Abstract

Dataset Versioning is extremely important for enterprises nowadays for ensuring reproducibility of results, tracking data changes over time, maintaining quality measures, enabling collaboration and ensuring legal compliance. Hence, cost-effective data management becomes necessary to reduce storage and reconstruction costs of datasets. In this work, we study the *cost efficient data versioning problem* where the goal is to optimize the storage and reconstruction costs of data versions, given a graph of datasets as nodes and edges capturing edit/reconstruction information. This problem (along with its variants) was introduced by Bhattacharjee et al. [1]. One central variant we study is MINSUM RETRIEVAL (MSR) where the goal is to minimize the total retrieval costs, while keeping the storage costs bounded. While such problems are frequently encountered in collaborative tools, e.g., version control of source code, data analysis pipelines etc., to the best of our knowledge, there is not much research studying the theoretical aspects of these problems.

We show the best known heuristic, LMG (introduced in [1]) can perform arbitrarily badly in certain cases. Moreover, we show that it is hard to get $o(n)$-approximation for MSR on general graphs even if we relax the storage constraints by an $O(\log n)$ factor. Similar hardness results are shown for other variants. We propose poly-time approximation schemes for tree-like graphs, motivated by the fact that the graphs arising in practice from typical edit operations are often not arbitrary. In fact, as version graphs typically have low treewidth, we further develop new algorithms for bounded treewidth graphs.

Furthermore, we propose two new heuristics and evaluate them empirically. First, we extend LMG by considering more potential "moves", to propose a new heuristic LMG-All. LMG-All consistently outperforms LMG while having comparable run time on a wide variety of datasets, i.e., version graphs. Secondly, we apply our tree algorithms on the minimum-storage arborescence of an instance, yielding algorithms that are qualitatively better than all previous heuristics for MSR, as well as for another variant BOUNDEDMIN RETRIEVAL (BMR).

Index Terms

component, formatting, style, styling, insert

I. INTRODUCTION

The management and storage of data versions has become increasingly important to enterprises. As an example, the increasing usage of online collaboration tools allows many collaborators to edit an original dataset simultaneously, producing multiple versions of datasets to be stored daily. Large number of dataset versions also occur often in industry data lakes [2] where huge tabular datasets like product catalogs might require a few records (or rows) to be modified periodically, resulting in a new version for each such modification. Furthermore, in Deep Learning pipelines, multiple versions are generated from the same original data for the purpose of training and insight generation. At the scale of terabytes or even petabytes, storing and managing all the versions is extremely costly in the aforementioned situations [3]. Therefore, it is no surprise that data version control is emerging as a hot area in the industry [4, 5, 6, 7, 8, 9], and even popular cloud solution providers like Databricks are now capturing data lineage information, which helps in effective data version management [10].

In a pioneering paper, Bhattacharjee et al. [1] proposed a model capturing the trade-off between *storage* cost and *retrieval* (recreation) cost. The problems they studied can be defined as follows. Given dataset versions and a subset of the *"deltas"* between them, 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 (i.e., store it explicitly) or we store a "delta" and rely on edit operations to retrieve the version from another materialized version if necessary. The downside of the latter is that, to retrieve a version that was not materialized, we have to incur a computational overhead that we call *retrieval cost*.

Figure 1, taken from Bhattacharjee et al. [1], illustrates the central point through different storage options. (i) shows the input graph, with annotated storage and retrieval costs . If the storage size is not a

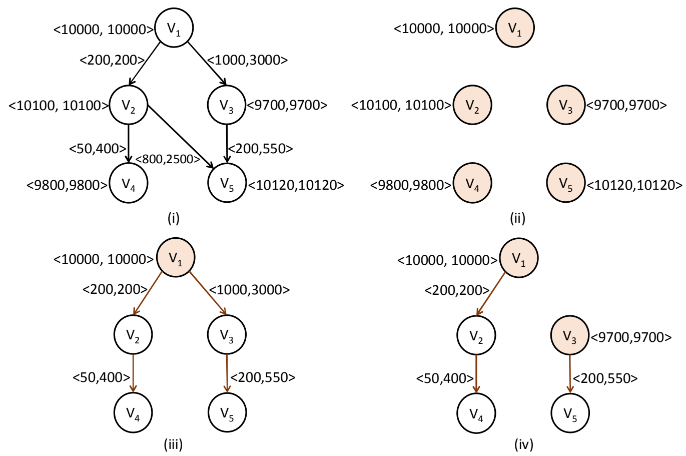

Fig. 1: (i) A version graph over 5 datasets – annotation ⟨a, b⟩ indicates a storage cost of a and a retrieval cost of b; (ii, iii, iv) three possible storage graphs. The figure is taken from [1]

concern, we should store all versions as in (ii). From (iii) to (iv), it is clear that, by materializing v_3, we shorten the retrieval costs of v_3 and v_5.

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 I).

Problem Name	Storage	Retrieval
MINIMUM SPANNING TREE	min	$\mathcal{R}(v) < \infty,\ \forall v$
SHORTEST PATH TREE	$< \infty$	min $\{\max_v R(v)\}$
MINSUM RETRIEVAL (MSR)	$\leq \mathcal{S}$	min$\{\sum_v R(v)\}$
MINMAX RETRIEVAL (MMR)	$\leq \mathcal{S}$	min $\{\max_v R(v)\}$
BOUNDEDSUM RETRIEVAL (BSR)	min	$\sum_v R(v) \leq \mathcal{R}$
BOUNDEDMAX RETRIEVAL (BMR)	min	$\max_v R(v) \leq \mathcal{R}$

TABLE I: Problems 1-6

There are some follow-up works on this model [11, 12, 13]. However, those either formulate new problems in different use cases [12, 13, 14] or implement a system incorporating the feature to store specific versions and deltas [13, 15, 16]. We will discuss this in more detail in Section I-B.

A. Our Contributions

We provide the first set of *approximation algorithms* and *inapproximability results* for the aforementioned optimization problems under various conditions which limit the shape of the underlying graph. In line with hardness results, in Section III we show that a simple path structure causes LMG, the best previous heuristic for MSR [1], perform arbitrary poorly.

MMR and BMR. In Section III we prove that it is hard to approximate MMR within $\log^* n$[4] factor and BMR within $\log n$ factor on general inputs. Meanwhile, in Section IV we give a polynomial-time dynamic programming (DP) algorithm for the two problems on *bidirectional trees*, i.e., digraphs whose underlying undirected graph is a tree. These inputs capture the cases where new versions are generated via edit operations.

[2]Both are assumptions in previous work [1] that simplify the problems. We note that our algorithms function even without these assumptions.
[3]This is true even if we relax $\mathcal{S}$ by $O(\log n)$.
[4]$\log^* n$ is "iterated logarithm", defined as the number of times we iteratively take logarithm before the result is at most 1.

Problem	Graph type	Assumptions	Inapproximability
MSR	arborescence	Triangle inequality $r = s$ on edges[1]	1
	undirected		$1 + \frac{1}{e} - \epsilon$
	general		$\Omega(n)$[2]
MMR	undirected		$2 - \epsilon$
	general		$\log^* n - \omega(1)$[3]
BSR	arborescence		1
	undirected		$(\frac{1}{2} - \epsilon) \log n$
BMR	undirected		$(1 - \epsilon) \log n$

TABLE II: Hardness results

Graphs	Problems	Algorithm	Approx.
General Digraph	MSR	LMG-All	heuristic
Bounded Treewidth	MSR & MMR	DP-BTW	$1 + \epsilon$
	BSR & BMR		$(1, 1 + \epsilon)$
Bidirectional Tree	MMR	DP-BMR	exact
	BMR		

TABLE III: Algorithms summary. Here, $\mathcal{R}_{max}$ is the maximum retrieval cost between any pair of vertices in the tree.

We also briefly describe an FPTAS (defined below) for MMR, analogous to the main result for MSR in Section V.

MSR and BSR. In Section III we prove that it is hard to design $\big(\Omega(n), \Omega(\log n)\big)$-bicriteria approximation[4] for MSR or $\Omega(\log n)$-approximation for MSR. It is also NP-hard to solve the two problems exactly on trees.

On the other hand, we again use DP to design a fully polynomial-time approximation scheme (FPTAS) for MSR on *bounded treewidth graphs*. These inputs capture many practical settings: bidirectional trees have width 1, series-parallel graphs have width 2, and the GitHub repositories we use in (Section VII) all have low treewidth. [5]

New Heuristics. We improved LMG into a more general LMG-All algorithm for solving MSR. LMG-All outperforms LMG in all our experiments and runs faster than LMG on sparse graphs.

Inspired by our algorithms on trees, we also propose two DP heuristics for MSR and BMR. Both algorithms perform extremely well even when the input graph is not tree-like. Moreover, there are known procedures for parallelizing general DP algorithms [17], so our new heuristics are potentially more practical than previous ones, which are all sequential.

B. Related Works

*1) **Theory**:* There was little theoretical analysis on the exact problems we study. The optimization problems are first formalized in Bhattacherjee et al. [1], which also compared the effectiveness of several proposed heuristics on both real-world and synthetic data. Zhang et al. [11] followed up by considering a new objective that is 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)* [18] and *Shallow-light Tree (SLT)* [19, 20, 21, 22, 23, 24]. However, this line of work focuses mainly on undirected graphs and their algorithms do not 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 our knowledge, there are only two works that give approximation algorithms for directed shallow-light trees. Chimani and Spoerhase [25] gives a bi-criteria

[4] An (α, β)-bicriteria pproximation refers to an algorithm that potentially exceeds the constraint by α times, in order to achieve a β-approximation of the objective. See Section II for an example.

[5] `datasharing`, `styleguide`, and `leetcode` have treewidth 2,3, and 6 respectively.

$(1+\epsilon, n^{\epsilon})$-approximation algorithm that runs in polynomial-time. Recently, Ghuge and Nagarajan [26] gave a $O(\frac{\log n}{\log\log n})$-approximation algorithm for *submodular tree orienteering* that runs in quasi-polynomial time. Their algorithm can be adapted into $O(\frac{\log^2 n}{\log\log n})$-approximation for BMR. For MSR, their algorithm gives $\big(O(\frac{\log^2 n}{\log\log n}), O(\frac{\log^2 n}{\log\log n})\big)$-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*.

*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 [27, 28]. 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 [29, 30, 31, 32, 33].

In the more flexible case, one may think of creating deltas without access to 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 as dataset discovery or dataset similarlity [2, 34, 35, 36, 37].

Several follow-up works of Bhattacherjee et al. [1] have implemented systems with a feature that saves only selected versions to reduce redundancy. There are works focusing on version control for relational databases [13, 15, 16, 38, 39, 40, 41, 42] and works focusing on graph snapshots [14, 43, 44]. However, since their focus was on designing full-fledged systems, the algorithms they proposed are rather simple heuristics with no theoretical guarantees.

*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. [1] that git's heuristic does not work well compared to other methods.

SVN, on the other hand, only stores the most recent version and the deltas to past versions [45]. Other existing data version management systems include [5, 6, 7, 8, 9], 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.

Though not directly related to our work, recently, there has been a lot of work exploring algorithmic and systems related optimizations for reducing storage and maintenance costs of data. For example, Mukherjee et al. [3] proposes optimal multi-tiering, compression and data partitioning, along with predicting access patterns for the same. Other works that exploit multi-tiering to optimize performance include e.g., [46, 47, 48, 49] and/or costs, e.g., [49, 50, 51, 52, 53, 54]. Storage and data placement in a workload aware manner, e.g., [49, 55, 56] and in a device aware manner, e.g., [57, 58, 59] have also been explored. [47] combine compression and multi-tiering for optimizing latency.

II. Preliminaries

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

A. Problem Setting

In the problems we study, we are given a directed *version graph* $G = (V, E)$, where vertices represent *versions* and edges capture the *deltas* between versions. Every edge e is associated with two weights: storage cost s_e and retrieval cost r_e.[8] The cost of storing e is s_e, and it takes r_e time to retrieve v once we retrieved u. Every vertex v is associated with only the storage cost, s_v, of storing (materializing) the

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

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

[8]If $e = (u, v)$, we may use $s_{u,v}$ in place of s_e and $r_{u,v}$ in place of r_e.

version. Since there is usually a smallest unit of cost in the real world, we will assume $s_v, s_e, r_e \in \mathbb{N}$ for all $v \in V, e \in E$.

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 is retrieved from materialized u with retrieval cost $R(v) = \sum_{i=1}^n r_{(u_{i-1},u_i)}$. 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 vertices $M \subseteq V$ and edges $F \subseteq E$ of *small* size (w.r.t. storage cost s), such that for each $v \in V \setminus M$, there is a *short* path (w.r.t retrieval cost r) from a materialized vertex to v. Formally, we want to minimize (a) total storage cost $\sum_{v \in M} s_v + \sum_{e \in F} s_e$, and (b) total (resp. maximum) retrieval cost $\sum_{v \in V} R(v)$ (resp. $\max_{v \in V} R(v)$).

Since the storage and retrieval objectives are negatively correlated, a natural problem is to constrain one objective and minimize the other. With this in mind, four different problems are formulated, as described by Problems 3-6 in Table I. These problems are originally defined in Bhattacherjee et al. [1], although we use different names for brevity. Since the first two problems are well studied, we do not discuss them further.

We note that MSR and BSR (MMR and BMR, resp.), are closely related. Given an algorithm for one, we can turn it into an algorithm for the other by binary-searching over the possible values of the constraint. Due to the somewhat exchangable nature of the storage and constraints in these problems, it's worth considering (α, β)-bicriteria approximations, where we relax the constraint by some α factor in order to achieve a β-approximation. For example, an algorithm A is (α, β)-bicriteria approximation for MSR if it outputs a feasible solution with storage cost $\leq \alpha \cdot \mathcal{S}$ and retrieval cost $\leq \beta \cdot OPT$ where OPT is the retrieval cost of an optimal solution.

B. *Further assumptions*

We hereby define several simplifications and complications that are primarily used Section III.

Triangle inequality: It is natural to assume that both weights satisfy triangle inequality, i.e., $r_{u,v} \leq r_{u,w} + r_{w,v}$, since we can always implement the delta $r_{u,v}$ by implementing first $r_{u,w}$ and then $r_{w,v}$. In fact, a more general triangle inequality should hold when we consider the materialization costs s_v, as it's often true that $s_u + s_{u,v} \geq s_v$ for all pairs of $u, v \in V$.

All hardness results in this paper hold under the generalized triangle inequality.

Directedness: It is possible that for two versions u and v, $r_{u,v} \neq r_{v,u}$. In real world, deletion is also significantly faster and easier to store than addition of content. Therefore, Bhattacherjee et al. [1] considered both directed and undirected cases; we argue that it is usually more natural to model the problems as directed graphs and focus on that case. Note that in the most general directed setting, it is possible that we are given the delta (u, v) but not (v, u). (or equivalently, $s_{v,u} \geq s_u$)

Single weight function: This is the special case where the storage cost function and retrieval cost function are identical. This can be seen in the real world, for example, when we use simple `diff` to produce deltas. We note all our hardness results hold for single weight functions. All our approximations hold for directed graphs with two weight functions.

Arborescence and trees: An *arborescence*, or a directed spanning tree, is a connected digraph where all vertices except a designated root have in-degree 1, and the root has in-degree 0. If each version is a modification on top of another version, then the "natural" deltas automatically form an arboreal input instance.[9] For practical reasons, we also consider *bi-directional tree* instances, meaning that both (u, v) and (v, u) are available deltas.[10] Empirical evidence shows that having deltas in both direction can greatly improve the quality of the optimal solution.

[9] This does not hold true for version controls because of the `merge` operation.

[10] While both edges are available, their storage costs and retrieval costs are not necessarily identical.

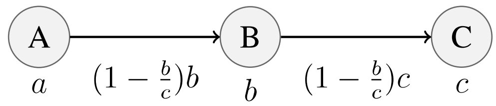

Fig. 2: An adversarial example for LMG.

Bounded treewidth: At a high level, treewidth measures how similar a graph is to a tree [60]. As one notable class of graphs with bounded treewidths, series-parallel graphs highly resemble the version graphs we derive from real-world repositories. Therefore, graphs with bounded treewidth is a natural consideration with high practical utility. We give precise definitions of this special case in Section V-C.

Non-uniform demand: Some versions may be requested more often than others.To model this, we may introduce *demands*d_v for $v \in V$, and replace total re-creation cost $\left(\sum_v R(v)\right)$ with *weighted* total re-creation cost $\left(\sum_v d_v R(v)\right)$ in MSR and BSR. This variant, although has great practical value, is not the focus of this paper. We demonstrate a hardness result when demand is non-uniform and hope to address this problem in future works.

III. Hardness results

We hereby prove the main hardness (inapproximability) results of the problems. For completeness, we define the notion of approximation algorithms, as used in this paper, in Appendix A. We also include in Appendix B a list of well-studied optimization problems that are used in this section for reduction purposes.

A. Heuristics can be Arbitrarily Bad

First, we consider the approximation factor of the best heuristic for MSR in Bhattacherjee et al. [1], 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

$$\rho = \frac{\text{reduction in total of retrieval costs}}{\text{increase in storage cost}}$$

. We provide the pseudo-code for LMG in Algorithm 1.

Note also that we work with the modified graph G_{aux} with the auxiliary root, as defined in Section II-A. 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.

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 is 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):

$$\rho_2 = \frac{(1-\epsilon)b + (1-\epsilon)c}{\epsilon c} = \frac{(1-\epsilon)b}{b} + \frac{1-\epsilon}{\epsilon} = \frac{1}{\epsilon} - \epsilon < \frac{2}{\epsilon} - 1$$

Algorithm 1: LOCAL MOVE GREEDY (LMG)

```
Input: Extended version graph G_aux, storage constraint 𝒮;
T ← minimum arborescence of G_aux rooted at v_aux w.r.t. weight function s;
Let S(T) be the total storage cost of T;
Let R(v) be the retrieval cost of v in T;
Let P(v) be the parent of v in T;
U ← V;
while S(T) < 𝒮 do
    (ρ_max, v_max) ← (0, ∅);
    for v ∈ U with S(T) + s_v − s_{P(v),v} ≤ 𝒮 do
        T' ← T \ {(P(v), v)} ∪ {(v_aux, v)};
        Δ = ∑_v (R(v) − R_{T'}(v));
        if Δ/(s_v − s_{P(v),v}) > ρ_max then
            ρ_max ← Δ/(s_v − s_{P(v),v});
            v_max ← v;
        end
    end
    T ← T \ {(P(v_max), v_max)} ∪ {(v_aux, v_max)};
    U ← U \ {v_max};
    if U = ∅ then
        return T;
    end
end
return T;
```

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. ■

B. Hardness Results on general graphs

Here, we show the various hardness of approximations on general input graphs. We first focus on MSR and MMR where the constraint is on storage cost and the objective is on the retrieval cost. We then shift our attention to BMR and BSR in which the constraint is of retrieval cost and the objective function is on minimizing storage cost.

1) Hardness for MINSUM RETRIEVAL *and* MINMAX RETRIEVAL*:*

Theorem 3.2: On version graphs with n nodes, even assuming single weight function and triangle inequality, there is no:

1) (α, β)-approximation for MINSUM RETRIEVAL if $\beta \leq \frac{1}{2}(1-\epsilon)\Big(\ln n - \ln \alpha - O(1)\Big)$; 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$(n^{O(\log\log n)})$;
2) $(1 + \frac{1}{e} - \epsilon)$-approximation for MINSUM RETRIEVAL on undirected graphs for all $\epsilon > 0$, unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$;
3) $\big(\log^*(n) - \omega(1)\big)$-approximation for MINMAX RETRIEVAL, unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$;
4) $(2-\epsilon)$-approximation for MINMAX RETRIEVAL on undirected graphs for all $\epsilon > 0$, unless NP =P.

Proof: **MinSum Retrieval.** There is an approximation-preserving (AP) reduction[11] from (Asymmetric) k-median to MSR. Let $s_{u,v} = r_{u,v} = d_{u,v}$, the distance from u to v in a (asymmetric) k-median instance. By setting the size of each version v to some large N and storage constraint to be $\mathcal{S} = kN + n$, we can restrict the instance to materialize at most k nodes and retrieve all other nodes through deltas. For large enough N, an (α, β)-approximation for MSR provides an (α, β)-approximation for (Asymmetric) k-median, just by outputting the materialized nodes. The desired results follow from known hardness for asymmetric [62] or symmetric (Appendix B) k-median.

MinMax Retrieval. A similar AP reduction exists from (Asymmetric) k-center to MMR. Again, we can set all materialization costs to N and $c_{u,v} = r_{u,v} = d_{u,v}$, and the desired result follows from the hardness of asymmetric [63] and symmetric [64] k-center. ■

2) Hardness for BoundedSum Retrieval *and* BoundedMax Retrieval*:*

Theorem 3.3: On both directed and undirected version graphs with n nodes, even assuming single weight function and triangle inequality, there is no:

1) $(c_1 \ln n)$-approximation for BoundedSum Retrieval for any $c_1 < 0.5$;
2) $(c_2 \ln n)$-approximation for BoundedMax Retrieval for any $c_2 < 1$.

unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$.

To prove this theorem, we will present our reduction to these two problems from Set Cover. We then show their structural properties on Theorems 3.4 and 3.5. We finally show the proof at the end of this section.

Reduction. Given a set cover instance with sets $A_1, \ldots, A_m$ and elements $o_1, \ldots, o_n$, we construct the following version graph:

1. Build versions a_i corresponding to A_i, and b_j corresponding to o_j. All versions have size N for some large $N \in \mathbb{N}$.
2. For all $i, j \in [m], i \neq j$, create symmetric delta (a_i, a_j) of weight 1. For each $o_j \in A_i$, create symmetric delta (a_i, b_j) of weight 1.

Lemma 3.4 (BMR's structure): Assume we are given an approximate solution to BMR on the above version graph under max retrieval constraint $\mathcal{R} = 1$. In polynomial time, we can produce another solution, of equivalent or better quality, such that:

1) Only the set versions are materialized. i.e., all $\{b_j\}_{j=1}^n$ are retrieved via deltas.
2) The storage cost does not exceed that of the original approximate solution, and the maximum retrieval cost is feasible.

Proof of Theorem 3.4: We show (1) by contradiction. Suppose the algorithm produces a solution that materializes b_j.

Case 1: If there exists a_i that needs to be retrieved through b_j (i.e., $o_j \in A_i$), then we can replace the materialization of b_j with that of a_i and replace edges of the form (b_j, a_k) with (a_i, a_k). It is straightforward to see that neither storage cost nor retrieval cost increased in this process.

Case 2: If no other node is dependent on b_j, we can pick any a_i such that (a_i, b_j) exists (again, $o_j \in A_i$). If a_i is already materialized in the original solution, then we can store (a_i, b_j) instead of materializing b_j, which decreases storage cost.

Case 3: If no a_i adjacent to b_j is materialized in the original solution, then some delta $(a_{i'}, a_i)$ has to be stored with the former materialized to satisfy the $\mathcal{R} = 1$ constraint. We can hence materialize a_i, delete the delta $(a_{i'}, a_i)$, and again replace the materialization of b_j with the delta (a_i, b_j) without increasing the storage. Figure 3 illustrates this case. ■

Lemma 3.5 (BSR's structure): Assume we are given an approximate solution to BSR on the above version graph under total retrieval constraint $\mathcal{R} = m - m_{\text{OPT}} + n$, where m_{OPT} is the size of the optimal set cover. In polynomial time, we can produce an improved solution such that:

1) Only the set versions are materialized. i.e., all $\{b_j\}_{j=1}^n$ are retrieved via deltas.

[11] See, e.g., Crescenzi [61] for more detail.

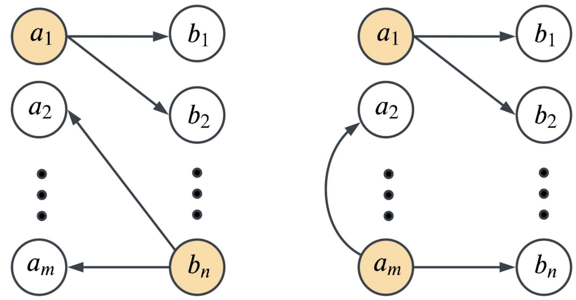

Fig. 3: Case 1 in proof of Theorem 3.4. The improved solution is on the right.

2) The storage cost does not exceed that of the original approximate solution, and the total retrieval cost is feasible.

Proof of Lemma 3.5:

We refer to the same three cases as in Theorem 3.4, and we want to show that, if b_j is materialized,

Case 1: if some a_i is retrieved through b_j, we can apply the same modification as Theorem 3.4. We can replace the materialization of b_j with that of a_i, and replace edges of the form (b_j, a_k) with (a_i, a_k). Neither the storage nor the retrieval cost increases in this case.

Now, WLOG we assume no deltas (b_j, a_i) are chosen.

Case 2: if no a_i is retrieved through b_j, and some a_i adjacent to b_j is materialized, then method in Theorem 3.4 needs to be modified a bit in order to remove the materialization of b_j. If we simply retrieve b_j via the delta (a_i, b_j), we would lower the storage cot by $N-1$ and *increase* the total retrieval cost by 1. This renders the solution infeasible if the total retrieval constraint is already tight.

To tackle this, we analyze the properties of the solutions with total retrieval cost exactly $\mathcal{R}$. Observe that all solutions must materialize at least m_{OPT} nodes at all time, so a configuration exhausting the constraint R must have some version w with retrieval cost at least 2. If this w is a set version, we can loosen the retrieval constraint by storing a delta of cost 1 from some materialized set instead. If w is an element version, then we can materialize its parent version (a set covering it), which increases storage cost by $N-1$ and decreases total retrieval cost by at least 2.

Either case, by performing the above action if necessary, we can resolve case 2 and obtain a approximate solution that is not worse than before.

Case 3: this is where each a_i adjacent to b_j neither retrieves through b_j nor is materialized. Fix an a_i, then some delta $(a_{i'}, a_i)$ has to be stored to retrieve a_i; WLOG we can assume that the former is materialized. We can thus materialize a_i, delete the delta $(a_{i'}, a_i)$, and again replace the materialization of b_j with the delta (a_i, b_j) with no increase in either costs. ∎

Equipped with Theorem 3.4 and Theorem 3.5, we are now ready to prove Theorem 3.3.

Proof of Theorem 3.3: Assumign $m = O(n)$ in the set cover instance, we present an AP reduction from SET COVER to both BMR and BSR.

BOUNDEDMAX RETRIEVAL. To produce a set cover solution, we take an improved approximate solution for BMR, and output the family of sets whose corresponding versions are materialized. Since none of the b_j's is stored, they have to be retrieved from some a_i. Moreover, under the constraint $\mathcal{R} = 1$, they have to be a 1-hop neighbor of some a_i, meaning the materialized a_i covers all of the elements in the set cover instance.

Finally, we prove that the approximation factor is preserved: for large N, the improved solution has objective value $\approx N|\{i : a_i \text{ materialized}\}|$. Hence, assuming $n = O(m)$, an $\alpha(|V|)$-approximation for MMR provides a $(\alpha(n) + O(1))$-approximation for set cover. Hence we ca not have $\alpha(|V|) = c \ln n$ for $c < 1$ unless NP $\subseteq$ DTIME$(n^{O(\log \log n)})$ [65].

BOUNDEDSUM RETRIEVAL. Assume for the moment that we know m_{OPT}, then we can set total retrieval constraint to be $\mathcal{R} = m - m_{\text{OPT}} + n$, and work with an improved approximate solution. This

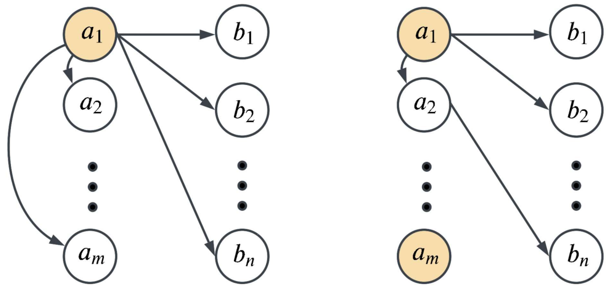

Fig. 4: The BSR case in proof of Theorem 3.3. The solution on the right has one version (b_2) of retrieval cost 2, hence it must materialize an additional version a_m to satisfy the total retrieval constraint.

choice of $\mathcal{R}$ is made so that an optimal solution must materialize the set versions corresponding to a minimum set cover. All other nodes must be retrieved via a single hop.

By Theorem 3.5, we assume all element versions are retrieved from a (not necessarily materialized) set version that covers it. If $m = O(n)$, an $\alpha(|V|)$-approximation of BMR materializes $m_{\text{ALG}} \leq (\alpha(n) + O(1))m_{\text{OPT}}$ nodes.

Note that, by materializing additional nodes, we are allowing a set B of b_j's to have retrieval cost ≥ 2. Let H denote the set of "hopped sets" A_i, which are not materialized yet are necessary to retrieve some b_j through the delta (a_i, b_j). By analyzing the total retrieval cost, we can bound $|H|$ by:

$$|H| \leq |B| \leq m_{\text{ALG}} - m_{\text{OPT}}$$

Specifically, each additional $b_j \in B$ increases retrieval cost by at least 1 compared to the optimal configuration; yet each of the $m_{\text{ALG}} - m_{\text{OPT}}$ additionally materialized set versions only decreases total retrieval cost by 1. It follows that the family of sets

$$S = \{A_i : a_i \text{ materialized }\} \cup H$$

is a $\Big(2\alpha(n) - O(1)\Big)$-approximation solution for the corresponding SET COVER instance. S is feasible because all of the b_j's are retrieved through some (a_i, b_j), where $A_i \in S$; on the other hand, the size of both sets on the right hand side are at most $(\alpha(n) + O(1))m_{\text{OPT}}$, hence the approximation factor holds. Thus, any $\alpha(|V|) = c \ln n$ for any $c < 0.5$ will result in a SET COVER approximation factor of $2c \cdot \ln(n)$.

We finish the proof by noting that, without knowing m_{OPT} in advance, we can run the above procedure for each possible guess of the value m_{OPT}, and obtaining a feasible set cover each iteration. The desired approximation factor is still preserved by outputting the minimum set cover solution over the guesses. ■

As a side note, MINSUM RETRIEVAL becomes impossibly hard on general graphs when non-uniform demands are allowed:

Theorem 3.6: On directed version graphs with $r = s$, triangle inequality, and non-uniform demand, MINSUM RETRIEVAL is inapproximable.

Proof: This follows from the same reduction from ASYMMETRIC K-MEDIAN as in Section III-B1. ■

C. Hardness on Arborescence

We show that MSR and BSR are NP-hard on arborescence instances. This essentially shows that our FPTAS algorithm for MSR in Section V-A is the best we can do in polynomial time.

Theorem 3.7: On arborescence inputs, MINSUM RETRIEVAL and BOUNDEDSUM RETRIEVAL are NP-hard even when we assume single weight function and triangle inequality.

In order to prove the theorem above, we rely on the following reduction which connects two problems together.

Lemma 3.8: If there exists poly-time algorithm $\mathcal{A}$ that solves BOUNDEDSUM RETRIEVAL (resp. BOUNDEDMAX RETRIEVAL) on some set of input instances, then there exists a poly-time algorithm solving MINSUM RETRIEVAL (resp. MINMAX RETRIEVAL) on the same set of input instances.

Proof: Suppose we want to solve a MSR (resp. MMR) instance with storage constraint $\mathcal{S}$. We can use $\mathcal{A}$ as a subroutine and conduct binary search for the minimum retrieval constraint $\mathcal{R}^*$ under which BSR (resp. BMR) has optimal objective at most $\mathcal{S}$. Thus, $\mathcal{R}^*$ is an optimal solution for our problem at hand.

To see that the binary search takes poly(n) steps, we note that the search space for the target retrieval constraint is bounded by $n^2 r_{max}$ for BSR and nr_{max} for BMR, where $r_{max} = \max_{e \in E} r_e$. ■

Now we show the proof for Theorem 3.7.

Proof of Theorem 3.7: Assuming Theorem 3.8, it suffices to show the NP-hardness of MSR on these inputs.

Consider an instance of SUBSET SUM problem with values $a_1, \ldots, a_n$ and target T. This problem can be reduced to MSR on an n-nary arborescence of depth one. Let the root version be v_0 and its children $v_1, \ldots, v_n$. The materialization cost of v_i is set to be $a_i + 1$ for $i \in [n]$, while that of v_0 is some N large enough so that the generalized triangle inequality holds. For each $i \in [n]$, we can set both retrieval and storage costs of edge (v_0, v_i) to be 1.

Consider MSR on this graph with storage constraint $\mathcal{S} = N + n + T$. From an optimal solution, we can construct set $A = \{i \in [n] : v_i \text{ materialized}\}$, an optimal solution for the above SUBSET SUM instance. ■

IV. EXACT ALGORITHM FOR MMR AND BMR ON BI-DIRECTIONAL TREES

As discussed in introduction, we can use an algorithm for BMR to solve for MMR via binary search. Hence, it suffices to focus on BMR, namely, when we are given maximal retrieval constraint $\mathcal{R}$ and want to minimize storage cost.

Algorithm 2: DP-BMR

Input: T, a tree, and $\mathcal{R}$, the max retrieval constraint;
Orient T arbitrarily. Sort V in reverse topological order;
DP$[v][u] \leftarrow \infty$ for all $v, u \in V$;
for v *in* V **do**
  for u *in* V *such that* $R(u, v) \leq \mathcal{R}$ **do**
    if $u = v$ **then**
      DP$[v][u] \leftarrow s_v$;
    else
      DP$[v][u] \leftarrow s_{p[v],v}$, where $p[v]$ is the node before v on the path from u to v;
    end
    for w *that is a child of* v **do**
      if w *in the path from* u *to* v **then**
        DP$[v][u] \leftarrow DP[v][u] +$ DP$[w][u]$;
      else
        DP$[v][u] \leftarrow DP[v][u] + \min\{OPT[w],$ DP$[w][u]\}$;
      end
    end
    $OPT[v] \leftarrow \min\{DP[v][w] : w \in V(T_{[v]})\}$;
  end
end
return OPT$[v_{root}]$;

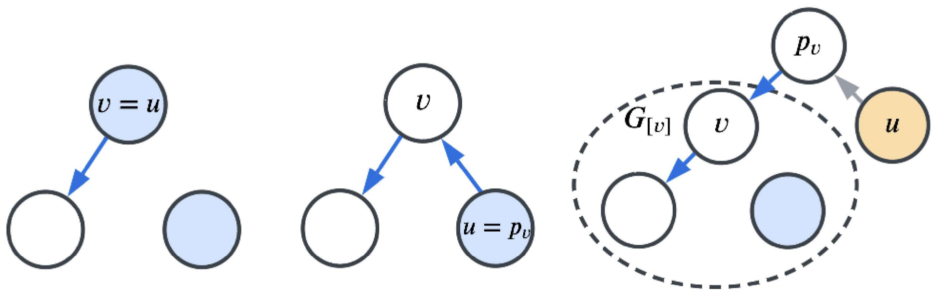

Fig. 5: 3 cases of DP-BMR, where $u = v, u \in V(T_{[v]})$, and $u \notin V(T_{[v]})$ respectively. The blue nodes and edges are stored in the partial solution.

Let $T = (V, E)$ be a bidirectional tree (abbreviated "tree") and let $\mathcal{R}$ be the maximum retrieval cost constraint. We can pick any vertex v_0 as root, and orient the tree such that v_0 has no parent, while all other nodes have exactly one parent.

For each $v \in V$, 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 $u, v \in V$, let $\text{DP}[v][u]$ be the minimum storage cost of a *partial solution* on $T_{[v]}$, which satisfies the following: all descendants of v are retrieved from some node in $T_{[v]}$ (possibly itself), while v is retrieved from the materialized version u, which is *potentially outside the subtree* $T_{[v]}$. See Figure 5 for an illustration.

Importantly, 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 $u - v$ path. For a leaf v, we set $DP[v][v] = s_v$, and $DP[v][u] = s_{(p_v^u, v)}$ for all $u \neq v$ with $R(u, v) \leq \mathcal{R}$. Here, p_v^u is just the 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 $\text{DP}[v][u] = \infty$ in these cases.

Recurrence. For convenience, we define 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 materialized descendants.*[12] In other words,

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

For recurrence on $\text{DP}[v][u]$ such that $R(v, u) \leq \mathcal{R}$, there are three possible cases of the relationship between v and u (see Figure 5). In each case, we outline what we add to $\text{DP}[v][u]$.

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

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 in each DP variable.

Theorem 4.1: BOUNDEDMAX RETRIEVAL is solvable on bidirectional tree instances in $O(n^2)$ time.

[12]Note that the case where v is retrieved from u outside of $T_{[v]}$, or case 3 in Figure 5, is not considered in this helper variable.

The time complexity follows from the observation that each calculation of $DP[v][u]$ in the recurrence takes $O(\deg(v))$ time, and $\sum_u \sum_v \deg(v) = \sum_u O(n) = O(n^2)$. The optimality of this DP can be shown inductively from leaves up, and is ommited due to space limitations.

We note that by binary-searching the constraint value $\mathcal{S}$, this algorithm also solves MINMAX RETRIEVAL on trees.

V. FPTAS FOR MSR VIA DYNAMIC PROGRAMMING

In this section we work on MINSUM RETRIEVAL and present a fully polynomial time approximation scheme (FPTAS) on digraphs whose *underlying undirected graph* has bounded treewidth. Similar techniques can be applied to MMR, but we will focus on MSR due to space constraints.

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

A. Warm-up: Bidirectional Trees

As a warm-up to the more general algorithm, we present an FPTAS for bidirectional tree instances of MSR via DP. This algorithm also inspired a practical heuristic DP-MSR, presented in Section VI-B.

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

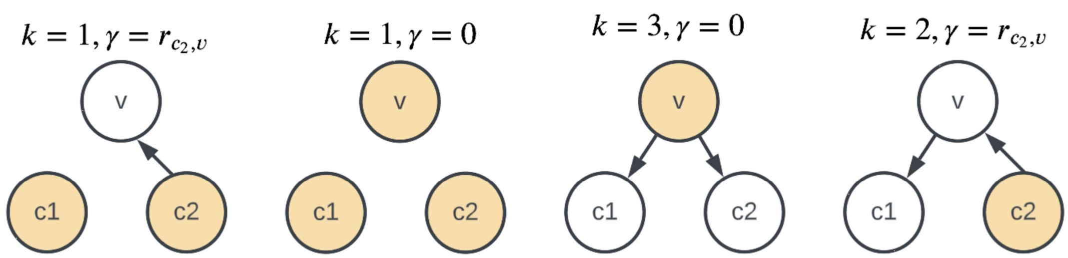

Fig. 6: An illustration of DP variables in Section V-A

DP variables. We define $\mathrm{DP}[v][k][\gamma][\rho]$ to be the minimum storage cost for the subproblem with constraints v, k, γ, ρ such that (with examples illustrated in Figure 6)

1) *Root for subproblem* $v \in V$ is a vertex on the tree; in each iteration, we consider the subtree rooted at v.
2) *Dependency number* $k \in \mathbb{N}$ is the number of versions retrieved from 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 the subtree root v, if it is retrieved from a materialized descendant. 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.

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:

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

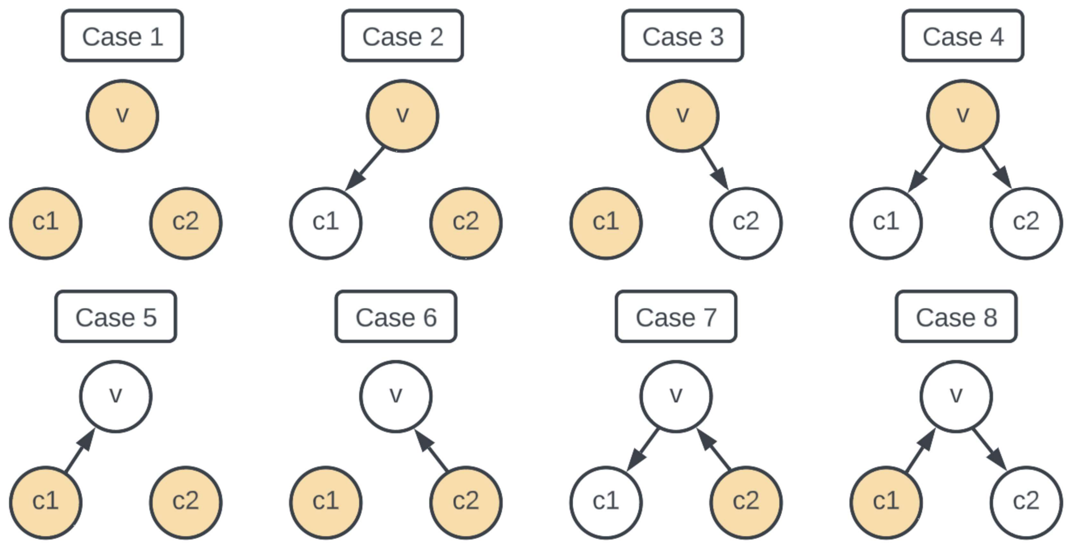

Fig. 7: Eight types of connections on a binary tree. A node is colored if it is materialized or retrieved via delta from outside the chart. Otherwise, an uncolored node is retrieved from another node as illustrated with the arrows.

and $T(\epsilon)$ is the number of "ticks" we want to partition the retrieval range into. The choice for $T(\epsilon)$ will be specified in the proof for Theorem 5.2. We will work with r' in the rest of the subsection. However, by an abuse of notation, we still use r for discretized retrieval for the ease of representation.

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

Recurrence step. On each iteration at node v, we take the minimum over all possible situations as illustrated in Figure 7. For the DP recurrence, we want to restrict the possible solutions illustrated in Figure 7 on v to the corresponding compatible partial solutions on c_1 and c_2. The recurrence relation for all cases is given in our full paper. Here, we select representative cases and explain the details of calculation below:

*1) **Dealing with dependency**:* When we decide to retrieve any child from v, like illustrated by case 4 of Figure 7, the children c_1, c_2 along with all their dependencies now become dependencies of v. The minimum storage cost in case 4 (given $v, k, \gamma = 0, \rho$) is:

$$S_4 = s_v + s_{v,c_1} + s_{v,c_2} - s_{c_1} - s_{c_2} \tag{1}$$

$$+ \min_{\substack{\rho_1+\rho_2=\rho \\ k_1+k_2=k-1}} \Big\{ DP[c_1][k_1][0][\rho_1 - k_1 r_{v,c_1}] \tag{2a}$$

$$+ DP[c_2][k_2][0][\rho_2 - k_2 r_{v,c_2}] \Big\} \tag{2b}$$

In Eq. (2a), 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 has dependency number k_1 and c_2 has that of k_2.

Also in Eq. (2a), the choice of ρ_1, ρ_2 determines how we are allocating retrieval costs budget ρ to c_1 and c_2 respectively. Specifically in Eq. (2a) and Eq. (2b), 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. Similar applies to the subproblem on $G_{[c_2]}$.

Next, we highlight the idea of "invisible" dependency here: for case 2 on $G_{[v]}$, note the diff (v, c_1) and (v, c_2) was not available in any previous recurrence, since v has just been introduced. Therefore, the compatible solution for the subproblems on $G_{[c_1]}$ and $G_{[c_2]}$ have to materialize nodes c_1 and c_2 to ensure they can be retrieved. This explains the $-s_{c_1} - s_{c_2}$ terms in Eq. (1), as when calculating the retrieval cost on $G_{[v]}$, we need to subtract back the additional materialization cost.

When generalizing the DP onto graphs with bounded treewith, similarly, restriction of a global solution does not always result in a feasible partial solution because of the existence of dependencies invisible to the subproblems. We resolve them using similarly ideas, as discussed in Section V-C.

*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 $v, k = 0, \gamma, \rho$,

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

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.

*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:

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

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 which achieves the following

$$\exists k \leq n, \gamma \in \mathbb{N} \quad \text{s.t.} \quad DP[v_{root}][k][\gamma][\rho] \leq \mathcal{S}$$

We shall formally state and prove the FPTAS result below.

Lemma 5.1: The DP algorithm outputs a configuration with total retrieval cost at most $\text{OPT} + \epsilon r_{max}$ in poly$(n, 1/\epsilon)$ time.

Proof: By setting $T(\epsilon) = \frac{n^4}{\epsilon}$, we have $l = \frac{n^2 r_{max}}{T(\epsilon)} = \frac{\epsilon r_{max}}{n^2}$. Note that we can get an approximation of the original retrieval costs by multiplying each r'_e with l. This creates an estimation error of at most l on each edge. Note further that in the optimal solution, at most n^2 edges are materialized, so if ρ^* is the minimal discretized total retrieval cost, we have

$$\text{total retrieval of output} \leq l\rho^* \leq \text{OPT} + n^2 l \leq \text{OPT} + \epsilon r_{max}.$$

■

Now we prove the main theorem of this subsection:

Theorem 5.2: For all $\epsilon > 0$, there is a $(1+\epsilon)$-approximation algorithm for MINSUM RETRIEVAL on bidirectional trees that runs in poly$(n, \frac{1}{\epsilon})$ time.

Proof: Given parameter ϵ, we can use the DP algorithm as a black box and iterate the following for up to n times:

1) Run the DP for the given ϵ on the current graph. Record the output.

2) Let (u, v) be the most retrieval cost-heavy edge. We now set $r_{(u,v)} = 0$ and $s_{(u,v)} = s_v$. If the new graph is infeasible for the given storage constraint, or if all edges have already been modified, exit the loop.

At the end, we output the best out of all recorded outputs. This improves the previous bound when $r_{max} > \text{OPT}$: at some point we will eventually have $r_{max} \leq \text{OPT}$, which means the output configuration, if mapped back to the original input, is a feasible $(1 + \epsilon)$-approximation. ■

B. 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 5.3 (Tree Decomposition [60]): A tree decomposition of undirected $G_0 = (V_0, E_0)$ 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 $v \in V_0$, the bags containing v induce a connected subtree of T;
3) For each $(u, v) \in E_0$, there exists $z \in V_T$ such that S_z contains both u and v.

The *width* of a tree decomposition $T = (V_T, E_T)$ is $\max_{z \in V_T} |S_z| - 1$. The *treewidth* of 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 [66], but it is not suitable for our purpose.

We will WLOG assume a special kind of decomposition:

Definition 5.4 (Nice Tree Decomposition [67]): A nice tree decomposition is a tree decomposition with a designated root, where each node z is one of the following types:

1) A **leaf**, which has no children and whose bag has size 1;
2) A **forget node**, which has one children c, and $S_z \subset S_c$ and $|S_c| = |S_z| + 1$.
3) An **introduce node**, which has one children c, and $S_z \supset S_c$ and $|S_c| + 1 = |S_z|$.
4) A **join**, which has children c_1, c_2, and $S_z = S_{c_1} = S_{c_2}$.

Given a bound k on the treewidth, there are multiple algorithms for calculating a tree decomposition of width k [68, 69, 70], or an approximation of k [71, 72, 73, 74].

For our case, the algorithm by Bodlaender [69] can be used to compute a tree decomposition in time $2^{O(k^3)} \cdot O(n)$, which is linear if the treewidth k is constant. Given 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 [67].

C. Generalized Dynamic Programming

Here we outline the DP for MSR on graphs whose underlying undirected graph G_0 has treewidth at most k.

*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. Before proceeding, let us define some additional notations. For any bag $z \in V_T$, let $T_{[z]}$ be the induced subtree of T rooted at z. We define $V_{[z]} = \bigcup_{z' \in V(T_z)} S_{z'}$ be the set of vertices in the bags of $T_{[z]}$, including S_z. Following that, $G_{[z]}$ is the induced subgraph of G by vertices $V_{[z]}$.

We now define the *DP states*. At a high level, each state describes some number of *partial solutions* on the subgraph induced by $V_{[z]}$, $G_{[z]}$. 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

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

and a natural number ρ_z:

(i) *Parent function* $\text{Par}_z : S_z \mapsto V_{[z]}$ describing the partial solution on $G_{[z]}$, restricted on S_z. If $\text{Par}_z(v) \neq v$ then v will be retrieved through the edge $(\text{Par}_z(v), v)$. If $\text{Par}_z(v) = v$ then v will be materialized.
(ii) *Dependency function* $\text{Dep}_z : S_z \mapsto [n]$. Similar to the dependency parameter in the warm-up, $\text{Dep}_z(v)$ counts the number of nodes in $V_{[z]}$ retrieved through v.
(iii) *Retrieval cost function* $\text{Ret}_z : S_z \mapsto \{0, \ldots, nr_{\max}\}$. Similar to the root retrieval parameter in the warm-up, $\text{Ret}_z(v)$ denotes the retrieval cost of version v in the partial solution on $G_{[z]}$.
(iv) *Ancestor function* $\text{Anc}_z : S_z \mapsto 2^{S_z}$. If $u \in \text{Anc}_z(v)$, then u is retrieved in order to retrieve v in this partial solution, i.e., v is dependent on u. We need this extra information to avoid directed cycles.
(v) ρ_z, the total retrieval cost of the subproblem according to the partial solution. Similar to its counterpart in the warm-up, all retrieval costs would be discretized by the same technique that makes the approximation an FPTAS.

A feasible state on $z \in V_T$ is a pair $(\mathcal{T}_z, \rho_z)$ which correctly describes some partial solution on $G_{[z]}$ 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 $G_{[z]}$.

We are now ready to describe how to compute the states.

*2) **Recurrence on leaves**:* For each leaf $z \in V_T$, the only feasible partial solution is to materialize the only vertex v in the leaf bag. We can easily calculate its state and storage cost.

*3) **Recurrence on forget nodes**:* This is also easy: for a forget node z with child c, we have $G_{[z]} = G_{[c]}$, and hence the states on z are simply the restrictions of states on c.

*4) **Recurrence on introduce nodes**:* At introduce node z with child c, we have $S_z = S_c \cup \{v_0\}$ for some "introduced" v_0. Each feasible state $(\mathcal{T}_z, \rho_z)$ on z must correspond to some state $(\mathcal{T}_c, \rho_c)$ on c, which we can calculate as follows:

We first initialize $\mathcal{T}_c$ to be the respective functions in $\mathcal{T}_z$ restricted on S_c. For instance, $\text{Par}_c = \text{Par}_z |_{S_c}$.

If v_0 is retrieved through $u \in S_c$ according to $\mathcal{T}_z$ ($\text{Par}_z(v_0) = u$), then we remove the dependencies related to v_0 and the retrieval cost incurred on edge (u, v_0). Specifically:

(i) Decrease the value of Dep_c by 1 on all vertices in $\text{Anc}_z(u)$.
(ii) Decrease ρ_c by $\text{Dep}_z(v_0) \cdot \text{Ret}_z(v_0)$.
(iii) Remove $\text{Anc}_z(u)$ from the ancestor functions of all descendants of z.

If v_0 has some child w according to $\mathcal{T}_z$ (namely, $\text{Par}_z(w) = v_0$), then we reverse the *uprooting* process in the warm-up, such that vertex w, which was not a root in $\mathcal{T}_z$, is now a root in $\mathcal{T}_c$. Specifically:

(i) Let $\text{Par}_c(w) = w$.
(ii) Remove v_0 from the ancestor function of w and all its descendants.
(iii) Decrease the retrieval cost function of w and its descendants by $\text{Ret}_z(w)$.
(iv) Decrease ρ_c by $\text{Ret}_z(w) \times \text{Dep}_z(w)$.

Since v could have multiple children, the last procedure is potentially repeated multiple times.

*5) **Recurrence on joins**:* Suppose we are at a join z with children a, b, where $S_z = S_a = S_b$. On a high level, for each state $(\mathcal{T}_z, \rho_z)$ on $G_{[z]}$, we want to find all pairs of states $(\mathcal{T}_a, \rho_a)$ and $(\mathcal{T}_b, \rho_b)$ such that the partial solutions they describe can combine into a partial solution on $G_{[z]}$, as described by $(\mathcal{T}_z, \rho_z)$. The pseudocode of the following functions can be found in the full version.

Compatibility. The algorithm COMPATIBILITY decides whether $\mathcal{T}_a, \mathcal{T}_b$ are indeed the "restrictions" of $\mathcal{T}_z$ on $G_{[a]}$ and $G_{[b]}$ respectively. If the algorithm returns true, we would later proceed to calculate the correct value of $\rho_a + \rho_b$, based on this particular restriction.

Resolving external retrieval. COMPATIBILITY first deals with the vertices that are retrieved from outside S_z. For example, each $v \in S_z$ retrieved from $V_{[a]} \setminus S_z$, like the yellow node in (c) of Figure 8, is instead materialized from $\mathcal{T}_b$'s perspective. To check whether $\mathcal{T}_a$ and $\mathcal{T}_b$ resolve all such cases correctly, we define subroutine EXTERNAL-RETRIEVAL to loop through S_z topologically and calculate the correct $\text{Par}, \text{Ret}, \text{Anc}$ functions for both $\mathcal{T}_a$ and $\mathcal{T}_b$.

Resolving external dependency. The next step in COMPATIBILITY is to check whether the functions $\text{Dep}_a, \text{Dep}_b$ are compatible with Dep_z. Specifically, nodes in S_z could have *external dependencies* in

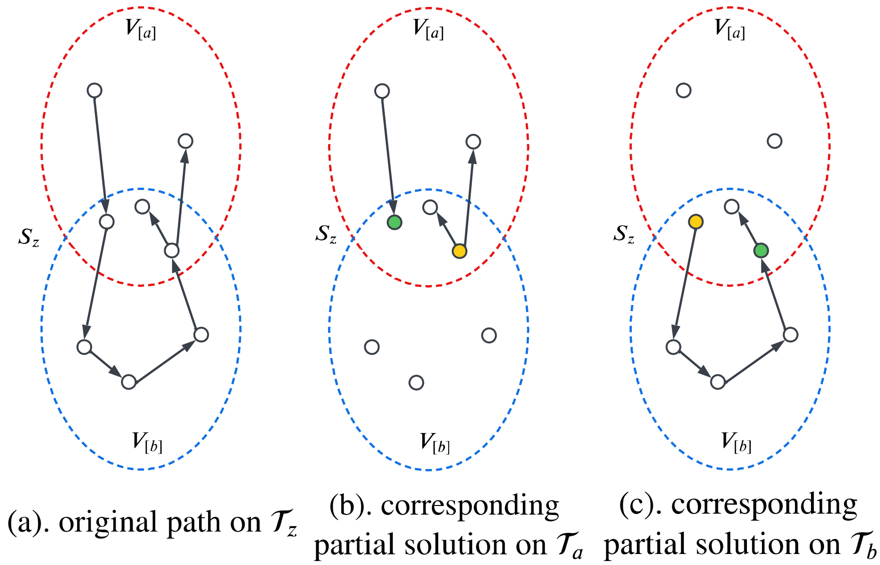

Fig. 8: Illustration for compatibility. Figure (b) and (c) show a pair of compatible configurations on $\mathcal{T}_a$ and $\mathcal{T}_b$ with the configuration on $\mathcal{T}_z$ in (a). The configurations of yellow nodes and green nodes are analyzed in EXTERNAL-RETRIEVAL and EXTERNAL-DEPENDENCY respectively.

$V_{[a]} \setminus S_z$ and $V_{[b]} \setminus S_z$, such as the green nodes in Figure 8 and Figure 9. The specific definition of $\mathrm{ExtDep}_a(v)$ is the number of descendants that v have outside S_z, to whom v is the *closest* ancestor in S_z, according to $\mathcal{T}_a$. To see an example, note that only four red nodes are counted towards $\mathrm{ExtDep}_a(A)$ in Figure 9. The functions ExtDep_b and ExtDep_z are defined similarly according to $\mathcal{T}_b$ and $\mathcal{T}_z$.

We note that $\mathrm{ExtDep}_a(v) + \mathrm{ExtDep}_b(v) = \mathrm{ExtDep}_z(v)$ for all $v \in S_z$ in order for $(\mathcal{T}_a, \mathcal{T}_b)$ to be compatible with $\mathcal{T}_z$. To check this, we call EXTERNAL-DEPENDENCY on $\mathcal{T}_z, \mathcal{T}_a, \mathcal{T}_b$ as a subroutine of COMPATIBILITY. We note that this is similar to distributing the dependency number k to the two children in case 4 of Figure 7.

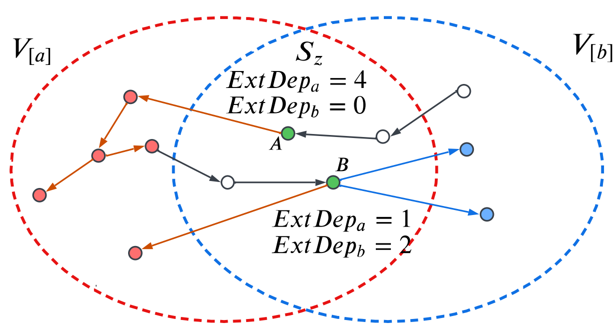

Fig. 9: Illustration for external dependency. Green nodes A and B both have non-zero external dependency, as labeled in the figure.

Calculating ρ. 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 cannot simply take $\rho_a + \rho_b = \rho_z$ due to the complicated procedure of combining $\mathcal{T}_a, \mathcal{T}_b$ into $\mathcal{T}_z$. We thus implement DISTRIBUTE RETRIEVAL to calculate ρ_Δ such that $\rho_a + \rho_b = \rho_z - \rho_\Delta$ and then iterate through all such ρ_a and ρ_b.

Recurrence relation. Finally, we have all we need for the recurrence relation. For each feasible $(\mathcal{T}_z, \rho_z)$, we take

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

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 where

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

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

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

Output The minimum retrieval cost of a global solution is just $\min\{\rho_z : \exists \mathcal{T}_z, \sigma(\mathcal{T}_z, \rho_z) \le \mathcal{S}\}$ over all feasible $(\mathcal{T}_z, \rho_z)$, where z is the designated root of the nice tree decomposition.

We conclude this section with the following theorem.

Theorem 5.5: For a constant $k \ge 1$, on the set of graphs whose undelying undirected graph has treewidth at most k, MINSUM RETRIEVAL admits an FPTAS.

To see that our algorithm above is an FPTAS for MSR, the proof is almost identical to the proof of Theorem 5.2 (Section V-A3) once we note that the number of partial solutions on each z is poly(n).

An FPTAS for MMR arises from a similar procedure. When the objective becomes the maximum retrieval cost, we can use ρ_z to represent the maximum retrieval cost in the partial solution. We then modify $\mathrm{Dep}_z(v)$ to represent the highest retrieval cost among all the nodes that are dependent on v. The recurrence relation is also changed accordingly. One can note that, like before, the new tuple $\mathcal{T}_z$ contains all the information we need for a subsolution on $G_{[z]}$.

The same algorithms extend to $(1, 1+\epsilon)$ bi-criteria approximation algorithms for BSR and BMR naturally, as the objective and constraint are reversed.

VI. HEURISTICS ON MSR AND BMR

In this section, we propose three new heuristics that are inspired by empirical observations and theoretical results.

A. LMG-All: Improvement over LMG

Here we provide a brief description of the greedy heuristic LMG [1] and the improved LMG-All. We refer to the full paper for pseudocodes and formal definitions.

On a high level, LMG does the following:

1) Find a configuration that minimizes total storage cost.
2) Let V_{active} be the set of vertices not yet materialized, and, if materialized, does not cause the configuration to exceed storage constraint $\mathcal{S}$. If $V_{active} = \varnothing$, output the current configuration.
3) For each $v \in V_{active}$, calculate the cost and benefit of materializing v: storage cost increases by some $S(v)$, but total retrieval cost decreases by some $R(v)$.
4) From all such v, materialize the one that maximizes $\frac{R(v)}{S(v)}$. Go to step 2 and repeat.

Our improved heuristic LMG-All enlarges the scope of the search on each greedy step. Instead of searching for the most efficient version to *materialize*, we explore the payoff of *modifying any single edge*:

1) Find a configuration that minimizes total storage cost.
2) Let Par(v) be the current parent of v on retrieval path. In addition to V_{active}, Define edge set E_{active} to be the edges that (a) does not cause the configuration to exceed storage constraint $\mathcal{S}$, and (b)

does not form cycles, if $(u, v) \in E_{active}$ were to replace $(\text{Par}(v), v)$ in the current configuration. If $V_{active} = E_{active} = \varnothing$, output the current configuration.

3) Calculate cost and benefit of each $v \in V_{active}$ and $e \in E_{active}$. Materialize or store the most cost-effective node or edge. Go to step 2 and repeat.

While LMG-All considers more edges than LMG, it is not obvious that LMG-All always provides a better solution, due to its greedy nature.

B. DP on extracted bi-directional trees

We propose DP heuristics on both MSR and BMR, as inspired by algorithms in Sections IV and V. To ensure a reasonable running time, we only run the DP's 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 V-A and Section IV) with input G'.

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.

VII. Experiments for MSR and BMR

In this section, we discuss the experimental setup and results for empirical validation of the algorithms' performance, as compared to previous best-performing heuristic: LMG for MSR, and MP for BMR.[13].

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

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

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% (to simulate decompression).

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

ER construction. Instead of the naturally constructing edges between each pair of parent and child commits, we construct the edges as in an Erdős-Rényi random graph: between each pair (u, v) of versions, with probability p both deltas (u, v) and (v, u) are constructed, and with probability $1 - p$ neither are constructed. The resulting graphs are much less tree-like.[14]

Dataset	#nodes	#edges	avg. storage cost of edge
datasharing	29	74	395
styleguide	493	1250	8659
996.ICU	3189	9210	337038
freeCodeCamp	31270	2.5×10^7	14800
LeetCode	246	628	1.2×10^7
LeetCode 0.05	246	3032	1.0×10^8
LeetCode 0.2	246	11932	1.0×10^8
LeetCode 1	246	60270	1.0×10^8

TABLE IV: Natural and ER graphs overview.

B. Results in MSR

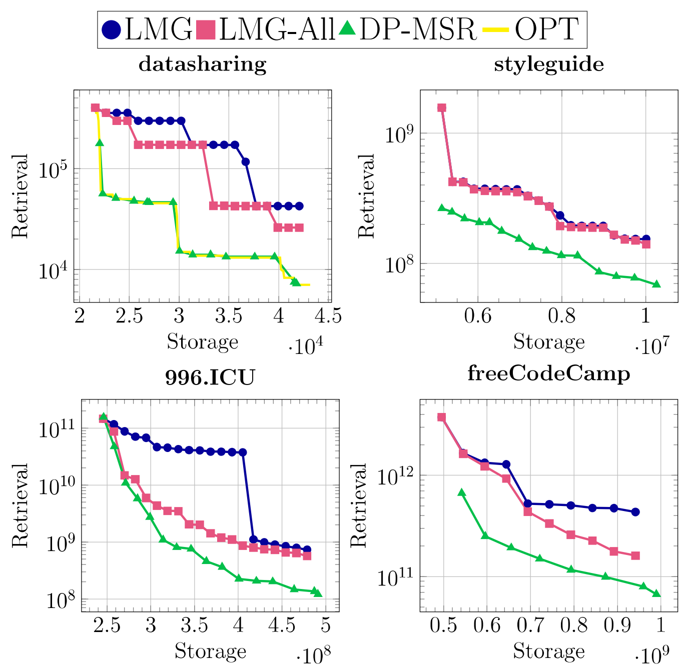

Fig. 10: Performance of MSR algorithms on natural graphs. OPT is obtained by solving an integer linear program (ILP) using Gurobi [76]. ILP takes too long to finish on all graphs except `datasharing`.

Figure 10, Figure 11, and Figure 12 demonstrate the performance of the three MSR algorithms on natural graphs, compressed natural graphs, and compressed ER graphs. The running times for the algorithms are shown in Figure 11 and Figure 12. Since run time for most non-ER graphs exhibit similar trends, many are omitted here due to space constraint. Also note that, since DP-MSR generates all data points in a single run, 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.

[14]ER graphs have treewidth $\Theta(n)$ with high probability if the number of edges per vertex is greater than a small constant [75].

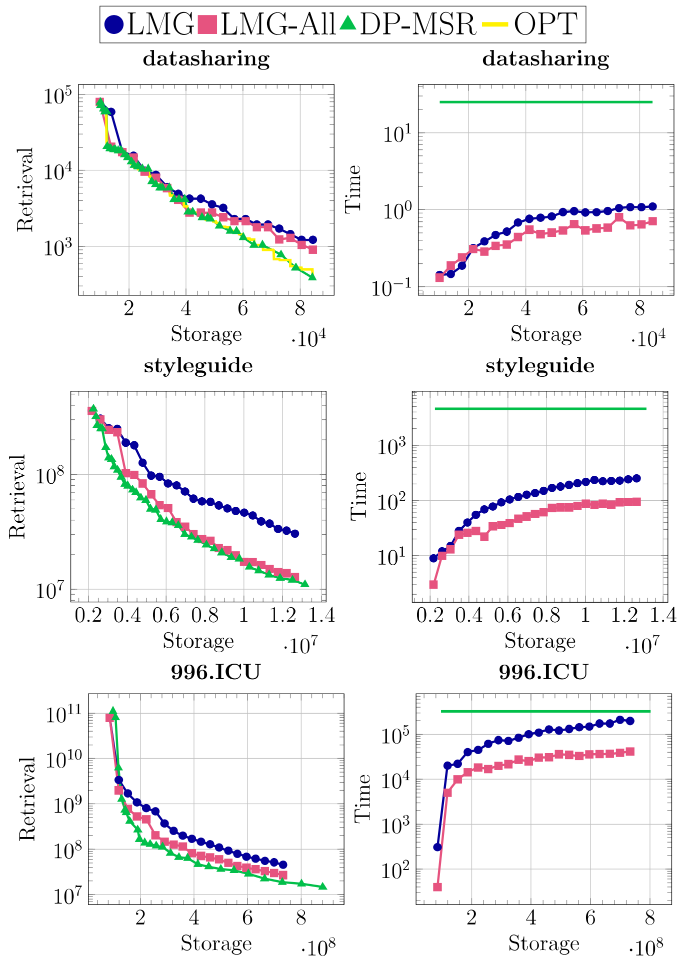

Fig. 11: Performance and run time of MSR algorithms on compressed graphs.

Performance analysis. On most graphs, DP-MSR outperforms LMG-All, which in turn outperforms LMG. This is especially clear on natural version graphs, where DP-MSR solutions are near 1000 times better than LMG solutions on `996.ICU`. in Figure 10. On `datasharing`, DP-MSR almost perfectly matches the optimal solution for all constraint ranges.

On naturally constructed graphs (Figure 10), LMG-All 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 random compression (Figure 11), 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 12).

Run time analysis. For all natural graphs, we observe that LMG-All uses no more time than LMG (as shown in Figure 11). Moreover, LMG-All 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 could be due to LMG making bigger, more expensive changes on each iteration (materializing a node with many dependencies, for instance) as compared to LMG-All.

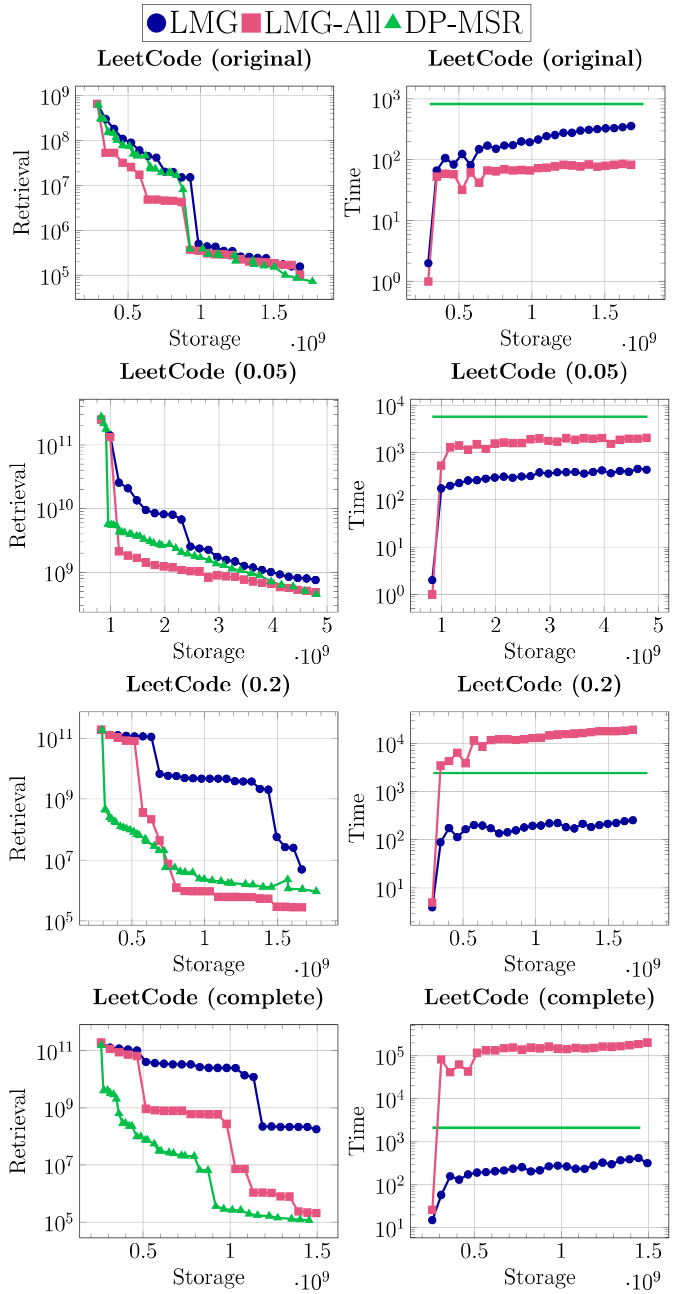

Fig. 12: Performance and run time of MSR algorithms on compressed ER graphs.

As expected, though, LMG-All takes much more time than the other two algorithms on denser ER graphs (Figure 12), 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 11). However, unlike LMG and LMG-All, the DP algorithm returns a whole spectrum of solutions at once, so it is difficult to make a direct comparison. We also note that the runtime of DP heavily depends on the choice of ϵ and the storage pruning bound. Hence, the user can trade-off the runtime with solution's quatlities by parameterize the algorithm with coarser configurations.

C. 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.

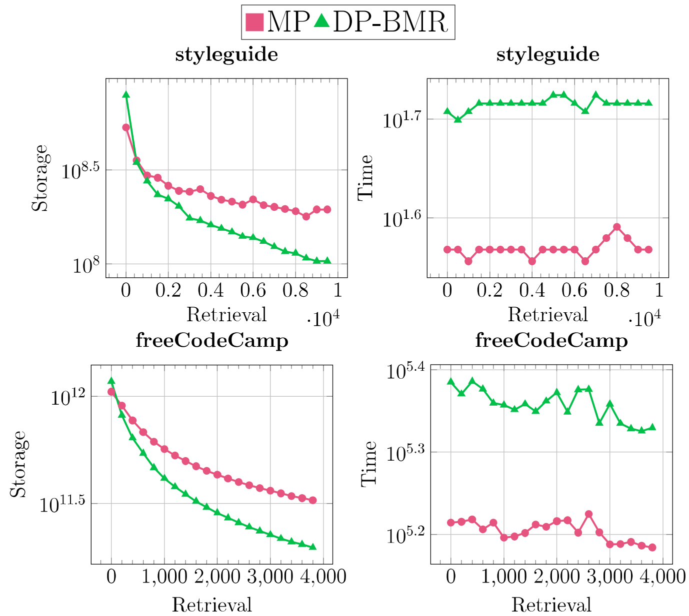

Fig. 13: Performance and run time of BMR algorithms on natural version graphs.

Due to space limitation, we only present the results on natural graphs, as shown in Figure 13, to respectively illustrate their performance and run time.

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 these are optimal solutions of the problem on the bidirectional tree.

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 13. Unlike LMG and LMG-All, their run times do not change much with varying constraint values.

Overall Evaluation For MSR, we recommend always using one of LMG-All and DP-MSR in place of LMG for practical use. On sparse graphs, LMG-All 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.

VIII. 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. For practical use, we extracted the idea behind this approach as well as previous LMG approach, and developed heuristics which significantly improved both the performance and run time in experiments, while potentially allowing for parallelization.

References

[1] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran, "Principles of dataset versioning: Exploring the recreation/storage tradeoff," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1346–1357, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1346-bhattacherjee.pdf

[2] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena, "Data lake management: Challenges and opportunities," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 1986–1989, aug 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352116

[3] K. Mukherjee, R. Shah, S. K. Saini, K. Singh, K. , H. Kesarwani, K. Barnwal, and A. Chauhan, "Towards optimizing storage costs on the cloud," *IEEE 39th International Conference on Data Engineering (ICDE) (To Appear)*, 2023.

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

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

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

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

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

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

[10] P. Roome, T. Feng, and S. Thakur, "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, 2022, last accessed: 13-Oct-22.

[11] Y. Zhang, H. Liu, C. Jin, and Y. Guo, "Storage and recreation trade-off for multi-version data management," in *Web and Big Data*, Y. Cai, Y. Ishikawa, and J. Xu, Eds. Cham: Springer International Publishing, 2018, pp. 394–409.

[12] B. Derakhshan, A. Rezaei Mahdiraji, Z. Kaoudi, T. Rabl, and V. Markl, "Materialization and reuse optimizations for production data science pipelines," ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1962–1976. [Online]. Available: https://doi.org/10.1145/3514221.3526186

[13] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. Parameswaran, "ORPHEUSDB: bolt-on versioning for relational databases (extended version)," *The VLDB Journal*, vol. 29, no. 1, pp. 509–538, 2020.

[14] N. N. Manne, S. Satpati, T. Malik, A. Bagchi, A. Gehani, and A. Chaudhary, "CHEX: multiversion replay with ordered checkpoints," *Proceedings of the VLDB Endowment*, vol. 15, no. 6, pp. 1297–1310, 2022.

[15] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "Forkbase: An efficient storage engine for blockchain and forkable applications," *Proc. VLDB Endow.*, vol. 11, no. 10, p. 1137–1150, jun 2018. [Online]. Available: https://doi.org/10.14778/3231751.3231762

[16] M. E. Schule, L. Karnowski, J. Schmeißer, B. Kleiner, A. Kemper, and T. Neumann, "Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB," *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, pp. 169–180, 2019.

[17] A. Stivala, P. J. Stuckey, M. G. de la Banda, M. Hermenegildo, and A. Wirth, "Lock-free parallel dynamic programming," *Journal of Parallel and Distributed Computing*, vol. 70, no. 8, pp. 839–848, 2010.

[18] S. Khuller, B. Raghavachari, and N. E. Young, "Balancing minimum spanning trees and shortest-path trees," *Algorithmica*, vol. 14, no. 4, pp. 305–321, 1995. [Online]. Available: https://doi.org/10.1007/BF01294129

[19] G. Kortsarz and D. Peleg, "Approximating shallow-light trees," in *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 1997, pp. 103–110.

[20] M. T. Hajiaghayi, G. Kortsarz, and M. R. Salavatipour, "Approximating buy-at-bulk and shallow-light k-steiner trees," *Algorithmica*, vol. 53, no. 1, pp. 89–103, 2009.

[21] B. Haeupler, D. E. Hershkowitz, and G. Zuzic, "Tree embeddings for hop-constrained network design," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*,

ser. STOC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 356–369. [Online]. Available: https://doi.org/10.1145/3406325.3451053

[22] M. V. Marathe, R. Ravi, R. Sundaram, S. Ravi, D. J. Rosenkrantz, and H. B. Hunt III, "Bicriteria network design problems," *Journal of algorithms*, vol. 28, no. 1, pp. 142–171, 1998.

[23] R. Ravi, "Rapid rumor ramification: approximating the minimum broadcast time," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 202–213.

[24] M. R. Khani and M. R. Salavatipour, "Improved approximations for buy-at-bulk and shallow-light k-steiner trees and (k,2)-subgraph," *J. Comb. Optim.*, vol. 31, no. 2, p. 669–685, feb 2016. [Online]. Available: https://doi.org/10.1007/s10878-014-9774-5

[25] M. Chimani and J. Spoerhase, "Network Design Problems with Bounded Distances via Shallow-Light Steiner Trees," in *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), E. W. Mayr and N. Ollinger, Eds., vol. 30. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 238–248. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/4917

[26] R. Ghuge and V. Nagarajan, "Quasi-polynomial algorithms for submodular tree orienteering and directed network design problems," *Mathematics of Operations Research*, vol. 47, no. 2, pp. 1612–1630, 2022.

[27] P. Buneman, S. Khanna, and W.-C. Tan, "Data Provenance: Some Basic Issues," *Lecture Notes in Computer Science*, pp. 87–93, 2000.

[28] Y. L. Simmhan, B. Plale, D. Gannon *et al.*, "A survey of data provenance techniques."

[29] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta algorithms: an empirical analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 192–214, 1998.

[30] R. C. Burns and D. D. E. Long, "In-place reconstruction of delta compressed files," *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing - PODC '98*, pp. 267–275, 1998.

[31] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Performance Evaluation*, vol. 79, pp. 258–272, 2014.

[32] J. MacDonald, "File system support for delta compression," Ph.D. dissertation, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkley, 2000.

[33] D. T. N. M. T. Suel, "zdelta: An efficient delta compression tool," 2002.

[34] Y. Jayawardana and S. Jayarathna, "DFS: A Dataset File System for Data Discovering Users," *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, vol. 00, pp. 355–356, 2019.

[35] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, "Aurum: A Data Discovery System," *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1001–1012, 2018.

[36] D. Brickley, M. Burgess, and N. Noy, "Google Dataset Search: Building a search engine for datasets in an open Web ecosystem," *The World Wide Web Conference*, pp. 1365–1375, 2019.

[37] A. Bogatu, A. A. A. Fernandes, N. W. Paton, and N. Konstantinou, "Dataset Discovery in Data Lakes," *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, vol. 00, pp. 709–720, 2020.

[38] J. Brown and N. Weber, "DSDB: An Open-Source System for Database Versioning & Curation," *2021 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, vol. 00, pp. 299–307, 2021.

[39] A. Chavan and A. Deshpande, "DEX: Query Execution in a Delta-based Storage System," *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 171–186, 2017.

[40] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande, "Decibel: The Relational Dataset Branching System," *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, vol. 9, no. 9, pp. 624–635, 2016.

[41] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran, "DataHub: Collaborative Data Science & Dataset Version Management at Scale,"

arXiv, 2014.
[42] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker, "Efficient Versioning for Scientific Array Databases," *2012 IEEE 28th International Conference on Data Engineering*, vol. 1, pp. 1013–1024, 2012.
[43] T. Ying, H. Chen, and H. Jin, "Pensieve: Skewness-aware version switching for efficient graph processing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 699–713. [Online]. Available: https://doi.org/10.1145/3318464.3380590
[44] U. Khurana and A. Deshpande, "Efficient Snapshot Retrieval over Historical Graph Data," *arXiv*, 2012, 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.
[45] W. Nagel, "Subversion: not just for code anymore," *Linux Journal*, vol. 2006, no. 143, p. 10, 2006.
[46] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 219–230.
[47] H. Devarajan, A. Kougkas, L. Logan, and X.-H. Sun, "Hcompress: Hierarchical data compression for multi-tiered storage environments," in *2020 IEEE IPDPS*. IEEE, 2020, pp. 557–566.
[48] H. Devarajan, A. Kougkas, and X.-H. Sun, "Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments," in *2020 IEEE IPDPS*. IEEE, 2020, pp. 62–72.
[49] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt, "Cast: Tiering storage for data analytics in the cloud," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 45–56.
[50] A. Erradi and Y. Mansouri, "Online cost optimization algorithms for tiered cloud storage services," *Journal of Systems and Software*, vol. 160, p. 110457, 2020.
[51] M. Liu, L. Pan, and S. Liu, "To transfer or not: An online cost optimization algorithm for using two-tier storage-as-a-service clouds," *IEEE Access*, vol. 7, pp. 94 263–94 275, 2019.
[52] ——, "Keep hot or go cold: A randomized online migration algorithm for cost optimization in staas clouds," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4563–4575, 2021.
[53] W. Si, L. Pan, and S. Liu, "A cost-driven online auto-scaling algorithm for web applications in cloud environments," *Knowledge-Based Systems*, vol. 244, p. 108523, 2022.
[54] R. Kinoshita, S. Imamura, L. Vogel, S. Kazama, and E. Yoshida, "Cost-performance evaluation of heterogeneous tierless storage management in a public cloud," in *2021 Ninth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2021, pp. 121–126.
[55] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Taming the cloud object storage with mos," in *Proceedings of the 10th Parallel Data Storage Workshop*, 2015, pp. 7–12.
[56] ——, "Mos: Workload-aware elasticity for cloud object stores," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 177–188.
[57] L. Vogel, V. Leis, A. van Renen, T. Neumann, S. Imamura, and A. Kemper, "Mosaic: a budget-conscious storage engine for relational database systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2662–2675, 2020.
[58] R. Lasch, T. Legler, N. May, B. Scheirle, and K.-U. Sattler, "Cost modelling for optimal data placement in heterogeneous main memory," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2867–2880, 2022.
[59] R. Lasch, R. Schulze, T. Legler, and K.-U. Sattler, "Workload-driven placement of column-store data structures on dram and nvm," in *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*, 2021, pp. 1–8.
[60] U. Bertelè and F. Brioschi, "On non-serial dynamic programming," *Journal of Combinatorial Theory, Series A*, vol. 14, no. 2, pp. 137–148, 1973.

[61] P. Crescenzi, "A short guide to approximation preserving reductions," in *Proceedings of Computational Complexity. Twelfth Annual IEEE Conference*, 1997, pp. 262–273.

[62] A. Archer, "Inapproximability of the asymmetric facility location and k-median problems," 2000.

[63] J. Chuzhoy, S. Guha, E. Halperin, S. Khanna, G. Kortsarz, R. Krauthgamer, and J. S. Naor, "Asymmetric k-center is log* n-hard to approximate," *J. ACM*, vol. 52, no. 4, p. 538–551, jul 2005. [Online]. Available: https://doi.org/10.1145/1082036.1082038

[64] T. F. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theoretical computer science*, vol. 38, pp. 293–306, 1985.

[65] U. Feige, "A threshold of ln n for approximating set cover," *J. ACM*, vol. 45, no. 4, p. 634–652, jul 1998. [Online]. Available: https://doi.org/10.1145/285055.285059

[66] T. Johnson, N. Robertson, P. Seymour, and R. Thomas, "Directed tree-width," *Journal of Combinatorial Theory. Series B*, vol. 82, no. 1, pp. 138–154, May 2001, 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.

[67] H. L. Bodlaender, "A partial k-arboretum of graphs with bounded treewidth," *Theoretical Computer Science*, vol. 209, no. 1, pp. 1–45, 1998.

[68] ——, "A linear time algorithm for finding tree-decompositions of small treewidth," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 226–234. [Online]. Available: https://doi.org/10.1145/167088.167161

[69] S. Arnborg, D. G. Corneil, and A. Proskurowski, "Complexity of finding embeddings in a k-tree," *SIAM Journal on Algebraic Discrete Methods*, vol. 8, no. 2, pp. 277–284, 1987. [Online]. Available: https://doi.org/10.1137/0608024

[70] F. V. Fomin, I. Todinca, and Y. Villanger, "Large induced subgraphs via triangulations and cmso," *SIAM Journal on Computing*, vol. 44, no. 1, pp. 54–87, 2015. [Online]. Available: https://doi.org/10.1137/140964801

[71] M. Belbasi and M. Fürer, "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*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 273–287. [Online]. Available: https://doi.org/10.1007/978-3-030-92681-6_23

[72] F. V. Fomin, D. Lokshtanov, S. Saurabh, M. Pilipczuk, and M. Wrochna, "Fully polynomial-time parameterized computations for graphs and matrices of low treewidth," *ACM Trans. Algorithms*, vol. 14, no. 3, jun 2018. [Online]. Available: https://doi.org/10.1145/3186898

[73] T. Korhonen, "A single-exponential time 2-approximation algorithm for treewidth," in *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, 2022, pp. 184–192.

[74] U. Feige, M. Hajiaghayi, and J. R. Lee, "Improved approximation algorithms for minimum-weight vertex separators," in *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 563–572. [Online]. Available: https://doi.org/10.1145/1060590.1060674

[75] Y. Gao, "Treewidth of erdős–rényi random graphs, random intersection graphs, and scale-free random graphs," *Discrete Applied Mathematics*, vol. 160, no. 4-5, pp. 566–578, 2012.

[76] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[77] O. H. Ibarra and C. E. Kim, "Fast approximation algorithms for the knapsack and sum of subset problems," *J. ACM*, vol. 22, no. 4, p. 463–468, oct 1975. [Online]. Available: https://doi.org/10.1145/321906.321909

[78] R. M. Karp, "The fast approximate solution of hard combinatorial problems," in *Proc. 6th South-*

Eastern Conf. Combinatorics, Graph Theory and Computing (Florida Atlantic U. 1975), 1975, pp. 15–31.
[79] G. Gens and Y. Levner, "Approximate algorithms for certain universal problems in scheduling theory," *Engineering Cybernetics*, vol. 16, no. 6, pp. 31–36, 1978.
[80] G. Gens and E. Levner, "A fast approximation algorithm for the subset-sum problem," *INFOR: Information Systems and Operational Research*, vol. 32, no. 3, pp. 143–148, 1994.
[81] H. Kellerer, R. Mansini, U. Pferschy, and M. G. Speranza, "An efficient fully polynomial approximation scheme for the subset-sum problem," *Journal of Computer and System Sciences*, vol. 66, no. 2, pp. 349–370, 2003.
[82] K. Jain, M. Mahdian, and A. Saberi, "A new greedy approach for facility location problems," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 731–740. [Online]. Available: https://doi.org/10.1145/509907.510012
[83] R. Solis-Oba, *Approximation Algorithms for the k-Median Problem*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 292–320. [Online]. Available: https://doi.org/10.1007/11671541_10
[84] A. Archer, "Two o(log*k)-approximation algorithms for the asymmetric k-center problem," in *Integer Programming and Combinatorial Optimization*, K. Aardal and B. Gerards, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–14.

APPENDIX

A. Approximation algorithms

We hereby define the notion of approximation algorithms used in this paper.

ρ-approximation algorithm Let $\mathcal{P}$ be a minimization problem where we want to come up with a feasible solution x satisfying some constraints (e.g., $a \cdot x \leq b$). We say that an algorithm $\mathcal{A}$ is a ρ-approximation algorithm for $\mathcal{P}$ if $x_{\mathcal{A}}$, the solution produced by $\mathcal{A}$ is feasible and that $OPT \leq f(x_{\mathcal{A}}) \leq \rho \cdot OPT$ where OPT is an optimal objective value and $f(x)$ is the objective value of a solution x. Here, ρ is the *approximation ratio*. Generally, we want $\mathcal{A}$ to run in polynomial time.

Polynomial-time approximation scheme (PTAS) A polynomial-time approximation scheme is an algorithm $\mathcal{A}$ that, when given any fixed $\epsilon > 0$, can produce an $(1+\epsilon)$-approximation in time that is polynomial in the instance size. We say that $\mathcal{A}$ is a *fully polynomial-time approximation scheme (FPTAS)* if the runtime of $\mathcal{A}$ is polynomial in both the instance size and $1/\epsilon$.

Bi-criteria approximation In problems such as ours where optimizing an objective function while meeting all constraints is challenging, we can consider relaxing both aspects. We say that an algorithm $\mathcal{A}$ (α, β)-approximates problem $\mathcal{P}$ if the objective value of its output is at most α times the objective value of an optimal solution **and** the constraints are violated at most β times.[15]

B. Optimizations problems with known hardness

We hereby define a few problems with known hardness results that reduce to one of the versioning problems.

Before we show our hardness results, it is useful to introduce several other NP-hard problems to reduce from.

*Definition A.1 (*SET COVER*):* Elements $U = \{o_1, \ldots, o_n\}$ and subsets $S_1, \ldots, S_m \subseteq U$ are given. The goal is to find $A \subseteq [m]$ with minimum cardinality such that $\bigcup_{i \in A} S_i = U$.
SET COVER has no $c \ln n$-approximation for any $c < 1$, unless NP $\subseteq$ DTIME$(n^{O(\log \log n)})$ [65].

*Definition A.2 (*SUBSET SUM*):* Given real values $a_1, \ldots, a_n$ and a target value T. The goal is to find $A \subseteq [n]$ such that $\sum_{i \in A} a_i$ is maximized but not greater than T.
SUBSET SUM is also NP-hard, but its FPTAS is well studied [77, 78, 79, 80, 81].

[15]We allow $x \leq \beta y$ if the constraint $x \leq y$ is presented.

*Definition A.3 (*K-MEDIAN *and* ASYMMETRIC K-MEDIAN*):* Given nodes $V = \{1, \ldots, n\}$, k, and symmetric (resp. asymmetric) distance measures $D_{i,j}$ for $i, j \in V$ that satisfies triangle inequality. The goal is to find a set of nodes $A \subseteq V$ of cardinality at most k that minimizes

$$\sum_{v \in V} \min_{c \in A} D_{v,c}.$$

The symmetric problem is well studied. The best known approximation lower bound for this problem is $1 + \frac{1}{e}$. We note that an inapproximability result of $1 + \frac{2}{e}$ [82] is often mistakenly quoted for this problem, whereas the authors actually studied the k-median variant where the "facilities" and "clients" are in different sets. With the same method we can only get the hardness of $1 + 1/e$ in our definition.

The asymmetric counterpart is rarely studied. The manuscript [62] showed that there is no (α, β)-approximation (β is the relaxation factor on k) if $\beta \leq \frac{1}{2}(1 - \epsilon)(\ln n - \ln \alpha - O(1))$, unless NP $\subseteq$ DTIME$(n^{O(\log \log n)})$.

Notably, even symmetric $k-$median is inapproximable when triangle inequality is not assumed on the distance measure D. [83] However, this hardness is not preserved by the standard reduction to MSR (as in Section III-B1), since the path distance on graphs inherently satisfies triangle inequality.

*Definition A.4 (*K-CENTER *and* ASYMMETRIC K-CENTER*):* Given nodes $V = \{1, \ldots, n\}$, k, and asymmetric distance measures $D_{i,j}$ for $i, j \in V$ that satisfies triangle inequality. The goal is to find a set of nodes $A \subseteq V$ of cardinality at most k that minimizes

$$\max_{v \in V} \min_{c \in A} D_{v,c}.$$

The symmetric problem has a greedy 2-approximation, which is optimal unless P= NP [64].

The asymmetric variant has $\log^* k$-approximation algorithms [84], and one cannot get a better approximation than $\log^* n$ unless NP $\subseteq$ DTIME$(n^{O(\log \log n)})$, if we allow k to be arbitrary [63].

C. *Reduction from general tree to binary tree*

Lemma A.5: If algorithm $\mathcal{A}$ solves BMR on binary tree instances in $O(f(n))$ time where n is the number of vertices in the tree, then there exists algorithm $\mathcal{A}'$ solving BMR on all tree instances in $O(f(2n))$ time.

Proof Sketch: If a node v has more than two children, we modify the graph as follows:

1) Create node v' and attach it as a child of v.
2) Move all but the left-most children of v to be children of v'
3) Set the deltas of $(v, v') = (v', v) = 0$; set $(v', c_i) = (v, c_i)$ and $(c_i, v') = (c_i, v)$ for all transferred children c_i.

By repeating this process we obtain a binary tree with $\leq 2n$ nodes which has the same optimal objective value as before. Hence, after producing a binary tree, we can utilize the algorithm for binary tree to solve BMR on any tree. ■

D. *All connection cases for DP for* MSR *on trees*

We present the 5 cases in the recurrent step here as promised in Section V-A. All other cases are symmetric to the cases we present, hence omitted. We use S_i to denote the minimum storage cost in case i, as shown in Figure 7.

E. *Algorithms in Section V-C*

We present the pseudo code for Algorithms **??**, 3, and 5 below, as mentioned in Section V-C:

$$
\begin{aligned}
S_1 &= s_v && + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k_1,\gamma_1}\{DP[c_1][k_1][\gamma_2][\rho_1]\} \\
& && + \min_{k_2,\gamma_2}\{DP[c_2][k_2][\gamma_2][\rho_2]\Big\} \\
S_2 &= s_v + s_{v,c_1} - s_{c_1} && + \min_{\rho_1 \le \rho} \Big\{ DP[c_1][k-1][0][\rho_1 - (k-1)r_{v,c_1}] \\
& && + \min_{k',\gamma_2}\{DP[c_2][k'][\gamma_2][\rho-\rho_1]\}\Big\} \\
S_3 &= s_v + s_{v,c_2} - s_{c_2} && + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k',\gamma_1}\{DP[c_1][k'][\gamma_1][\rho_1]\} \\
& && + DP[c_2][k-1][0][\rho_2 - (k-1)r_{v,c_2}]\Big\} \\
S_4 &= s_v + s_{v,c_1} - s_{c_1} + s_{v,c_2} - s_{c_2} && + \min_{\rho_1+\rho_2=\rho} \min_{k_1+k_2=k-1} \Big\{ DP[c_1][k_1][0][\rho_1 - k_1 r_{v,c_1}] \\
& && + DP[c_2][k_2][0][\rho_2 - k_2 r_{v,c_2}]\Big\} \\
S_5 &= s_{c_1,v} && + \min_{\rho_1 \le \rho} \Big\{ \min_{k_1}\{DP[c_1][k_1][\gamma - r_{c_1,v}][\rho_1-\gamma]\} \\
& && + \min_{k_2,\gamma'}\{DP[c_2][k_2][\gamma'][\rho-\rho_1]\}\Big\} \\
S_6 &= s_{c_2,v} && + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k_2}\{DP[c_2][k_2][\gamma - r_{c_2,v}][\rho_2-\gamma]\} \\
& && + \min_{k_1,\gamma'}\{DP[c_1][k_1][\gamma'][\rho_1]\}\Big\} \\
S_7 &= s_{c_2,v} + s_{v,c_1} - s_{c_1} && + \min_{\rho_1+\rho_2=\rho} \Big\{ DP[c_1][k-1][0][\rho_1 - (k-1)\cdot(r_{v,c_1}+\gamma)] \\
& && + \min_{k'}\{DP[c_2][k'][\gamma - r_{c_2,v}][\rho_2-\gamma]\}\Big\} \\
S_8 &= s_{c_1,v} + s_{v,c_2} - s_{c_2} && + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k'}\{DP[c_1][k'][\gamma - r_{c_1,v}][\rho_1-\gamma]\} \\
& && + DP[c_2][k-1][0][\rho_2 - (k-1)\cdot(r_2+\gamma)]\Big\}
\end{aligned}
$$

F. *Calculation of ρ*

We hereby demonstrate that the method for calculating ρ_Δ in Algorithm 6 is indeed correct.

For a pair of compatible partial solutions $\mathcal{T}_a, \mathcal{T}_b$ with regards to $\mathcal{T}_z$, ρ_Δ is defined such that $\rho_a + \rho_b = \rho_z - \rho_\Delta$. Therefore, as we go down a path described by $\mathcal{T}_z$ in topological order, we analyze how many times the retrieval cost of an edge is counted by both ρ_a and ρ_b as compared to that by ρ_z. For example, in figure 14, the retrieval cost of edge $(1, 2)$ is counted 8 times in $\mathcal{T}_z$, zero times in $\mathcal{T}_a$, and twice in $\mathcal{T}_b$. The details are as below:

1) We observe that all edges in $\mathcal{T}_a$ and $\mathcal{T}_b$ are must also be in $\mathcal{T}_z$. Hence, it suffices to focus on all edges of $\mathcal{T}_z$
2) For each v not materialized in $\mathcal{T}$, we use the temporary variable Count to denote how many times the edge $e = (\text{Par}_z(v), v)$ is over/undercounted in ρ_z.
   To put this formally, we can abuse notation and let $\text{Dep}_z(e)$ be the number of times r_e is counted towards total retrieval cost in $\mathcal{T}_z$. Then we have

$$\text{Count} = \text{Dep}_z(e) - (\text{Dep}_a(e) + \text{Dep}_b(e))$$

Algorithm 3: EXTERNAL-RETRIEVAL

```
Input: S_z, T_z;
Let T'_a = T'_b = T_z;
Sort S_z in topological order according to Anc_z;
for v ∈ S_z do
    /* Resolving external ancestors from a.                        */
    if Par_z(v) ∈ V_[a] \ S_z then
        Par'_b(v) = v;
        for w ∈ S_z with w ≠ v and v ∈ Anc'_b(w) do
            Ret'_b(w) −= Ret'_b(v);
            Anc'_b(w) ← Anc'_b(w) \ Anc'_b(v);
        Ret'_b(v) ← 0;
        Anc'_b(v) ← ∅;
    /* Resolving external ancestors from b.                        */
    if Par_z(v) ∈ V_[b] \ S_z then
        Par'_a(v) = v;
        for w ∈ S)z with w ≠ v and v ∈ Anc'_a(w) do
            Ret'_a(w) −= Ret'_a(v);
            Anc'_a(w) ← Anc'_a(w) \ Anc'_a(v);
        Ret'_a(v) ← 0;
        Anc'_a(v) ← ∅;
return T'_a, T'_b;
```

Algorithm 4: EXTERNAL-DEPENDENCY

```
Input: S, T;
Sort S in topological order according to Anc;
for v ∈ S do
    Let ExtDep(v) = Dep(v) − Σ_{w∈S:Par(w)=v} Dep(w);
for v ∈ S do
    if Par(v) ∉ S then
        for u ∈ Anc(v) with u ≠ v do
            ExtDep(u) −= ExtDep(v)
return ExtDep;
```

where if $\text{Par}_a(v) \neq \text{Par}_z(v)$, clearly $\text{Dep}_a(e)$ should be 0, since it is not even stored in $\mathcal{T}_a$.

3) If both endpoints of e are in S_z, then the amount of retrieval cost overcount in ρ_z is exactly $\text{Count} \cdot r_e$. On the other hand, if e is a delta from outside S_z, the overcount should be $\text{Count} \cdot \text{Ret}_z(v)$, since the entire retrieval cost of v is overcounted Count times.

G. ILP Formulation

In the following formulation, we have integer variables $\{x_e\}$ representing how many $v \in V$ is retrieved through the edge e. I_e is a Boolean variable denoting whether edge e is stored. We work on the extend graph with the auxiliary node v_{aux} for convenience.

Algorithm 5: COMPATIBILITY

Input: $S_z, \mathcal{T}_z, \mathcal{T}_a, \mathcal{T}_b$;

```
/* EXTERNAL-RETRIEVAL returns the ``true restrictions" of the Par,
   Anc, and Ret functions.                                          */
```
$\mathcal{T}_a', \mathcal{T}_b' \leftarrow$EXTERNAL-RETRIEVAL$(S_z, \mathcal{T}_z)$;
if *$\mathcal{T}_a'$ disagree with $\mathcal{T}_a$ or $\mathcal{T}_b'$ disagree with $\mathcal{T}_b$ on functions* Par, Anc, *or* Ret **then**
  return False;
```
/* for each v ∈ S_z, EXTERNAL_DEPENDENCY returns the dependency of v
   that are outside of S_z.                                         */
```
$\text{ExtDep}_z \leftarrow$ EXTERNAL_DEPENDENCY $(S_z, \mathcal{T}_z)$;
$\text{ExtDep}_a \leftarrow$ EXTERNAL_DEPENDENCY $(S_z, \mathcal{T}_a)$;
$\text{ExtDep}_b \leftarrow$ EXTERNAL_DEPENDENCY $(S_z, \mathcal{T}_b)$;
if $\text{ExtDep}_z \neq \text{ExtDep}_a + \text{ExtDep}_b$ **then**
  return False;
return True;

Algorithm 6: DISTRIBUTE RETRIEVAL

Input: $S_z, \mathcal{T}_z, \rho_z, S_a, S_b, \mathcal{T}_a, \mathcal{T}_b$;

```
/* We want ρ_z = ρ_a + ρ_b + ρ_Δ:                                   */
```
$\rho_\Delta \leftarrow 0$;
for $v \in S_z$ *such that* $\text{Par}_z(v) \neq v$ **do**
  ```
  /* The number of times r_{Par_z(v),v} is counted towards ρ_z, minus the
     number of times it is counted towards ρ_a and ρ_b:              */
  ```
  Count $\leftarrow \text{Dep}_z(v)$;
  if $\text{Par}_a(v) = \text{Par}_z(v)$ **then**
    Count $-= \text{Dep}_a(v)$;
  if $\text{Par}_b(v) = \text{Par}_z(v)$ **then**
    Count $-= \text{Dep}_b(v)$;
  if $\text{Par}_z(v) \in S_z$ **then**
    ```
    /* The edge r_{Par_z(v),v} is over/undercounted:                 */
    ```
    $\rho_\Delta \leftarrow \rho_\Delta + \text{Count} \cdot r_{\text{Par}_z(v),v}$;
  else
    ```
    /* The entire Ret_z(v) is over/undercounted:                     */
    ```
    $\rho_\Delta \leftarrow \rho_\Delta + \text{Count} \cdot \text{Ret}_z(v)$;
return ρ_Δ;

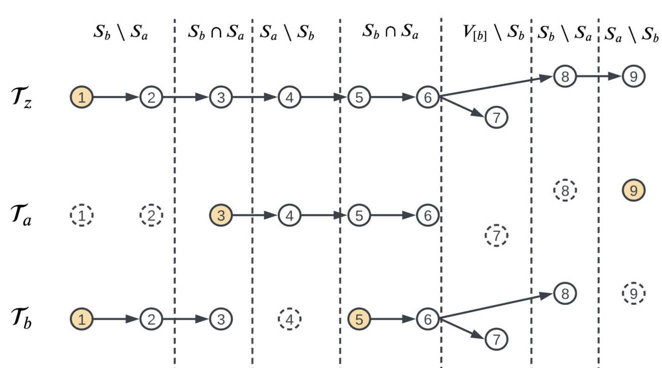

Fig. 14: Illustration of the retrieval path for Figure 8

$$
\begin{aligned}
\min \quad & \sum_{e \in E} r_e x_e && && \text{s.t.} \\
& x_e \leq |V-1| I_e && && \text{(indicator constraint)} \\
& \sum_{e \in E} s_e I_e && \leq \mathcal{R} && \text{(storage cost)} \\
& \sum_{e \in In(u)} x_e && = \sum_{e \in Out(u)} x_e + 1 \quad \forall u \in V \setminus \{v_{aux}\} && \text{(sink)} \\
& x_e && \in \{0, 1, \ldots, |V|\} && \\
& I_e && \in \{0, 1\} &&
\end{aligned}
$$