Efficient Parameterized Algorithms for Data Packing

KRISHNENDU CHATTERJEE, IST Austria, Austria
AMIR KAFSHDAR GOHARSHADY, IST Austria, Austria
NASTARAN OKATI, Ferdowsi University of Mashhad, Iran
ANDREAS PAVLOGIANNIS, EPFL, Switzerland

There is a huge gap between the speeds of modern caches and main memories, and therefore cache misses account for a considerable loss of efficiency in programs. The predominant technique to address this issue has been Data Packing: data elements that are frequently accessed within time proximity are packed into the same cache block, thereby minimizing accesses to the main memory. We consider the algorithmic problem of Data Packing on a two-level memory system. Given a reference sequence $R$ of accesses to data elements, the task is to partition the elements into cache blocks such that the number of cache misses on $R$ is minimized.

The problem is notoriously difficult: it is NP-hard even when the cache has size 1, and is hard to approximate for any cache size larger than 4. Therefore, all existing techniques for Data Packing are based on heuristics and lack theoretical guarantees.

In this work, we present the first positive theoretical results for Data Packing, along with new and stronger negative results. We consider the problem under the lens of the underlying access hypergraphs, which are hypergraphs of affinities between the data elements, where the order of an access hypergraph corresponds to the size of the affinity group. We study the problem parameterized by the treewidth of access hypergraphs, which is a standard notion in graph theory to measure the closeness of a graph to a tree. Our main results are as follows: we show that there is a number $q^*$ depending on the cache parameters such that (a) if the access hypergraph of order $q^*$ has constant treewidth, then there is a linear-time algorithm for Data Packing; (b) the Data Packing problem remains NP-hard even if the access hypergraph of order $q^* - 1$ has constant treewidth. Thus, we establish a fine-grained dichotomy depending on a single parameter, namely, the highest order among access hypergraphs that have constant treewidth; and establish the optimal value $q^*$ of this parameter.

Finally, we present an experimental evaluation of a prototype implementation of our algorithm. Our results demonstrate that, in practice, access hypergraphs of many commonly-used algorithms have small treewidth. We compare our approach with several state-of-the-art heuristic-based algorithms and show that our algorithm leads to significantly fewer cache-misses.

CCS Concepts: • Theory of computation → Fixed parameter tractability; • Software and its engineering → Compilers;

Additional Key Words and Phrases: compilers, data packing, cache management, data locality

ACM Reference Format:

*A full version of this paper is available at [Chatterjee et al. 2019] (https://repository.ist.ac.at/1056/). We are very thankful to the reviewers for their insightful comments. The research was partially supported by Vienna Science and Technology Fund (WWTF) Project ICT15-003, Austrian Science Fund (FWF) NFN Grant No S11407-N23 (RiSE/SHiNE), ERC Starting Grant (279307: Graph Games), and the IBM PhD Fellowship program.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/1-ART53
https://doi.org/10.1145/3290366
1 INTRODUCTION

We consider the problem of Data Packing over a two-level memory system consisting of a small cache and a large main memory. Given a reference sequence of memory accesses to data elements, the goal is to organize the data elements into blocks in order to minimize cache misses. Intuitively, putting contemporaneously-accessed elements in the same block reduces the number of cache misses.

**Cache Management.** Consider a memory system with an associative cache and a main memory. Data items are stored in the main memory and organized into sets of a small size, which are called blocks (or pages). All data items have the same size and all blocks can hold the same number of data items. The cache has a small capacity and can hold a few blocks at any given time. Whenever a program needs to access a data element, its corresponding block must be present in the cache before the access can happen. Therefore, if the block is not already in the cache, it will be copied into the cache from the main memory, potentially by evicting another block. This copying process is called a cache miss, and given the considerably slower speed of the main memory, cache misses are very time-consuming and lead to significant overhead [Wulf and McKee 1995]. Therefore, the problem of cache management, i.e., minimizing the number of cache misses, is of great importance in compilers and operating systems. Cache management can naturally be divided in two parts [Calder et al. 1998]: (i) deciding on how to replace the blocks in the cache, i.e. which block to evict when the cache is full and a miss occurs and (ii) deciding on the placement scheme of the data items inside blocks. These problems are respectively called Paging (or choosing a replacement policy) [Sleator and Tarjan 1985] and Data Packing [Lavaee 2016; Thabit 1982].

**Paging (Replacement Policy).** In paging, given a data placement scheme that divides the data items into blocks and a so-called reference sequence of accesses to data elements, the problem is to choose a block to be evicted each time a cache miss occurs. The goal is to do this in a way that minimizes the total number of cache misses over the reference sequence [Panagiotou and Souza 2006]. An algorithm that chooses the block to be evicted is called a replacement policy. Common replacement policies include FIFO, which evicts the oldest block in the cache, and LRU, which evicts the least recently used block [Borodin et al. 1995; Lavaee 2016]. Note that both FIFO and LRU can also be applied in the online setting, i.e., when the algorithm does not know the entire sequence in advance and can only observe accesses as they are made. In the offline case, where the entire reference sequence is given in the beginning, the optimal replacement policy is to evict the block whose first use is furthest in the future [Borodin et al. 1995]. This is called the optimal offline policy (OOP). We primarily focus on LRU as the replacement policy, because it is the one that is most commonly used in practice [Zhong et al. 2004].

**Data Packing.** The other aspect of cache management, which is the focus of this paper, is Data Packing [Thabit 1982]. Consider a cache with a capacity of \( m \) blocks, where each block can store \( p \) data items. Given a reference sequence \( R \) of length \( N \) of accesses to \( n \) distinct data items and a replacement policy, Data Packing asks for the optimal placement of data items into blocks in order to minimize the number of cache misses. The parameters \( m \) and \( p \) are considered to be small constants, and the complexity is studied wrt \( n \) and \( N \), which are large. Data Packing is an extremely hard problem and is known to be hard to approximate within any non-trivial factor, i.e., any factor significantly less than \( N \), unless \( P=NP \) [Lavaee 2016].

---

*This necessarily means that there is a limit on the size of data items and a large item should be broken into smaller parts, each of which is considered a distinct data item.*

---

Relevance of Data Packing in Programming Languages. Data Packing is an important technique for performance optimization during compilation and has been widely studied by the programming languages community (see Calder et al. 1998; Ding and Kennedy 1999; Lavaee 2016; Petrank and Rawitz 2002; Zhang et al. 2006; Zhong et al. 2004). The key relevance is two-fold:

- **Limit studies**: To test the performance of a compiler for data placement, various inputs are generated as benchmarks, and the baseline comparison of the performance is performed against an optimal algorithm [Petrank and Rawitz 2002]. Hence, an optimal data packing algorithm is necessary as the baseline.
- **Profiling**: Programs usually have similar memory-access behaviors over different inputs [Petrank and Rawitz 2002]. Hence, an effective approximate approach for the online cache management problem is to consider several representative inputs, then run an optimal offline algorithm for profiling, and then synthesize an answer to the online problem from optimal offline solutions [Calder et al. 1998; Petrank and Rawitz 2002].

Previous theoretical results on data packing have all been negative (hardness) results.

Heuristics and Affinity. Given the hardness of cache management and Data Packing, the research in this area has been mostly focused on developing heuristics. The intuition behind many of these heuristics is to exploit the underlying affinities between data elements by trying to place elements that are commonly accessed together in the same block or evicting the block that is less frequently accessed in conjunction with the rest of the blocks in the cache [Calder et al. 1998; Ding and Kennedy 1999; Ding and Kandemir 2014; Han and Tseng 2006; Zhong et al. 2004]. Some works, such as [Zhang et al. 2006], provide more sophisticated heuristics and construct a hierarchy of affinities. However, none of the existing heuristics provide any theoretical guarantees.

Access Graphs. The concept of access graphs [Borodin et al. 1995] has been introduced to model the affinities between data elements or blocks. An access graph is simply a graph in which there is a vertex corresponding to every data item and two vertices are connected by an edge if their respective items appear consecutively in the reference sequence. Access graphs might be weighted to model how many times every pair of elements have appeared consecutively. Similar structures and extensions of access graphs to access hypergraphs have been introduced in [Lavaee 2016; Thabit 1982] where they are called proximity (hyper)graphs. Moreover, most of the heuristic-based approaches also consider variants of the notion of access graphs.

Cache Misses vs Cache Hits. We consider the Data Packing problem, which asks to minimize the cache misses. Its natural dual problem is to maximize cache hits. While the two problems are equivalent in case of exact algorithms, an approximation algorithm for maximum cache hits does not necessarily lead to an approximation for minimum cache misses [Lavaee 2016]. For example, if in an access sequence of length $N$ we have $N - \sqrt{N}$ cache hits and $\sqrt{N}$ cache misses, an approximation of $N - \sqrt{N}$ hits can lead to an arbitrarily bad approximation of $\sqrt{N}$ cache misses. In practice, cache misses occur much less frequently than cache hits, but contribute significantly to the overhead. Thus, approximation of cache misses is more important than approximation of cache hits, and the Data Packing problem is defined in terms of cache misses.

Previous Results on Cache Management. To our knowledge, all theoretical results on minimizing cache misses are negative or hardness results. We summarize some of the main results in this area. Given a reference sequence $R$ of length $N$ and a cache with a capacity of $m$ blocks, the following results have been shown:

1. In [Petrank and Rawitz 2002], the authors considered the problem of Cache-conscious Data Placement, which is a somewhat different formulation of cache-miss minimization and is intuitively similar to Data Packing. In Cache-conscious Data Placement, a cache consists of $m$ lines, each capable of holding up to $p$ data items at any given time. The problem is to assign
each data item \(d\) to a cache line \(l_d\). When the program wants to access the data item \(d\), it should be present in cache line \(l_d\), otherwise a cache miss occurs and \(d\) is copied to \(l_d\), potentially by evicting another data item from \(l_d\). Given an eviction strategy, the goal is to assign data items to cache lines in a manner that minimizes cache misses. In [Petrank and Rawitz 2002] it was shown that the problem is NP-hard and unless \(P=NP\), it cannot even be approximated within a factor of \(O(N^{1-\epsilon})\).

(ii) In the same paper, it was shown that any algorithm that does not process the entire sequence, but instead relies on pairwise affinity information on data items, such as the access graph, cannot find a solution within a factor of \(m-3\) from the optimal, even with unbounded time.

(iii) In [Lavaee 2016], the author showed that Data Packing is NP-hard for any cache size and hard to approximate within a factor of \(O(N^{1-\epsilon})\) unless \(P = NP\).

Given these hardness results, Data Packing is usually handled by heuristic-based algorithms that do not provide any theoretical guarantee. The only positive theoretical result deals with approximating maximum cache hits:

(iv) In [Lavaee 2016] it was established that the dual problem of Data Packing with the goal of maximizing cache hits, instead of minimizing cache misses, is approximable within a constant factor. However, this does not approximate the optimal number of cache misses.

**Exploiting Structural Properties.** Data Packing is a notoriously difficult computational problem. In dealing with this problem, a direction that has not been pursued is to exploit structural properties of the access graphs. In many cases, structural properties of graphs help in obtaining efficient parametrized algorithms for computationally-hard problems. Specifically, a well-studied structural property in graph theory, which is frequently applied to computationally-hard graph problems, is the notion of treewidth. We present this notion below.

**Treewidth.** Treewidth [Robertson and Seymour 1984] is a well-known and extensively-studied parameter in graph theory. The treewidth of a graph is a measure of how tree-like the graph is. Specifically, trees and forests are the only graphs with a treewidth of 1. The importance of treewidth in algorithm design stems from the fact that many NP-hard graph problems (e.g., Vertex Cover and Hamiltonian Cycle) can be solved in polynomial time if the input graphs have constant treewidth and moreover, many other graph problems can be solved in a lower complexity [Abboud et al. 2016; Bodlaender 1997; Cygan et al. 2015; Fomin et al. 2017; Robertson and Seymour 1986]. The formal definition of treewidth is presented in the next section.

**Treewidth in Program Analysis.** Many important families of graphs that arise commonly in algorithm design are shown to have constant treewidth, e.g., series-parallel and outer-planar graphs [Bodlaender 1998]. Perhaps the most important example in program analysis is that the control-flow graphs of structured goto-free programs in many languages such as Pascal and C++ have constant treewidth [Thorup 1998]. The same result was also shown experimentally for most Java programs [Gustedt et al. 2002]. This has led to algorithmic advances in verification and program analysis [Chatterjee et al. 2015a]. Moreover, treewidth has also been exploited to obtain faster algorithms for static analysis of recursive state machines [Chatterjee et al. 2015b] and concurrent systems [Chatterjee et al. 2017, 2018].

**Treewidth in Data Packing.** In this work we show that Data Packing can be reduced to a graph problem. In many cases where a graph arises from a structured process, the treewidth of the graph is not very large [Bodlaender 1998]. For Data Packing, the access graphs arise from structured program accessing data from a well-defined data structure. Thus, it is natural to study the problem of Data Packing in terms of the treewidth property of the arising graphs.

**Our Contributions.** Our contributions include (a) polynomial algorithms for Data Packing in constant treewidth access (hyper)graphs, (b) stronger hardness results, and (c) experimental results
demonstrating that our approach leads to considerably fewer cache misses in comparison with previously-known heuristic-based approaches. Concretely, consider that the cache has size $m$, every block can hold $p$ data items and the reference sequence is of length $N$ with $n$ distinct items. We consider the access hypergraph of order $q$, where each vertex of the graph is a data item, and an edge connects a set of $q$ distinct data items if they appear contiguously in the reference sequence. We show that the Data Packing problem can be reduced to a graph partitioning problem of the access (hyper)graph and we study whether the constant treewidth property can be exploited for polynomial-time algorithms. Our main results, assuming constant $m$ and $p$, are as follows:

(1) **Results on Access Graphs.** We first consider $q = 2$. Note that order-2 access hypergraphs are basically access graphs. We establish the following results:

- **Linear-time algorithm.** We present a linear-time algorithm for Data Packing when the access graph is of constant treewidth and $m = 1$ (Theorem 3.1).
- **Hardness of the exact problem.** The Data Packing problem remains NP-hard for $m \geq 2$ and $p \geq 3$ even if the underlying access graph is a tree (which has treewidth 1) (Theorem 3.2).
- **Hardness of approximation.** Unless $P=NP$, for any $m \geq 6, p \geq 2$ and any constant $\epsilon > 0$, the Data Packing problem is hard to approximate within a factor of $O(N^{1-\epsilon})$ even if the underlying access graph is a tree (Theorem 3.2).

(2) **Results on Access Hypergraphs.** We define access hypergraphs of higher orders and consider their treewidth. Let $q^* = (m - 1)p + 2$. Note that $q^*$ depends only on the cache parameters, and not on $n$ or $N$. We consider the access hypergraph of order $q = q^*$. Intuitively, every edge of this access hypergraph contains all the necessary historical cache data for determining whether a miss occurs at a corresponding memory access under any given packing scheme. Formally, we establish the following results:

- **Linear-time algorithm.** We present a linear-time algorithm for Data Packing when the access hypergraph of order $q^*$ has constant treewidth (Theorem 4.1).
- **Hardness of the exact problem.** For $m \geq 2$ and $p \geq 3$, the Data Packing problem remains NP-hard even if the access hypergraph of order $q^* - 1$ has constant treewidth (Theorem 4.2).
• **Hardness of approximation.** Unless P=NP, for $m \geq 6$ and $p \geq 2$ and any constant $\epsilon > 0$, the Data Packing problem is hard to approximate within a factor of $O(N^{1-\epsilon})$ even if the access hypergraph of order $q^* - 4p - 1$ has constant treewidth (Theorem 4.3). Note that while constant treewidth has been exploited to obtain polynomial-time algorithms for NP-complete graph problems such as Vertex Cover and Hamiltonian Cycle, we show that for Data Packing the constant treewidth property does not always help, and the problem remains hard even when the access hypergraph of order $q^* - 1$ has constant treewidth.

Our hardness result and linear-time algorithm present a sharp boundary (or fine-grained dichotomy) that shows when the treewidth can be exploited. Concretely, the hardness of the Data Packing problem can be captured by a single parameter, namely, the highest order amongst access hypergraphs that have constant treewidth. We establish the optimal value $q^*$ of this parameter which is the necessary and sufficient condition for existence of efficient parameterized algorithms that exploit treewidth.

3 Experimental results. We present an experimental evaluation of a prototype implementation of our algorithm on a variety of benchmarks from linear algebra, sorting algorithms, dynamic programming, recursive algorithms, string matching, computational geometry and algorithms on tree data-structures. Our results show that the access hypergraphs of most of the benchmarks have small treewidth. We compare our approach with several state-of-the-art heuristic-based algorithms. The experimental results show that on average our optimal algorithms obtain 15-30% improvement over the previous heuristic-based approaches.

**Novelty and Significance.** In this paper, we define a novel and rich structural property of programs, i.e. access hypergraphs and their treewidth, and show that it can be exploited to obtain faster algorithms for Data Packing. Our algorithm is the first to produce optimal data packing schemes and is applicable to a large family of common programs. We also enrich the complexity landscape as shown in Figure 1. Only the results of Theorem 2.1 were known before, and all other results (which are shown in bold) are established in the present work.

2 PRELIMINARIES

2.1 Data Packing

In this section, we define the problem of data packing and fix our notation. We also present several previously-known results. The problem was first studied in [Thabit 1982]. Here, we present an adaptation of its definition as formalized in [Lavaeae 2016].

**Notation.** We use $\mathbb{Z}$ to denote the set of integers and $\mathbb{N}$ to denote the set of positive integers. Let $G = (V, E)$ be a (hyper)graph, and $X \subseteq V$. then we denote by $G[X]$, the induced subgraph of $G$ over $X$, i.e. $G[X] = (X, \{e \in E \mid e \subseteq X\})$. Given two (hyper)graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we define their union and intersection in the natural way, i.e. $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$ and $G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$. If $\mathcal{F}$ is a family of sets, we write $\cup \mathcal{F}$ (resp. $\cap \mathcal{F}$) to denote $\cup_{A \in \mathcal{F}} A$ (resp. $\cap_{A \in \mathcal{F}} A$). Given two functions $f, g : A \rightarrow \mathbb{Z}$, equality and summation are defined in a pointwise manner, i.e. $f \equiv g \Leftrightarrow \forall a \in A ; f(a) = g(a)$ and for any $a \in A$, we have $(f + g)(a) = f(a) + g(a)$. Given a function $f : A \rightarrow B$ and a subset $A' \subseteq A$, we use $f|_{A'}$ to denote the restriction of $f$ to $A'$. This restriction is a function of the form $f|_{A'} : A' \rightarrow B$ that agrees with $f$ on every point in $A'$. For a set $X$, we write $P(X)$ to denote the power set of $X$, i.e., the set of all subsets of $X$.

**Data Placement Schemes.** Given a set $D$ of size $n$ of data items and a positive integer $p$, a data placement scheme $\sigma$ is a partitioning of $D$ into blocks of size at most $p$. We call $p$ the **packing factor**. It is often useful to think of $\sigma$ as an equivalence relation on $D$ whose equivalence classes are the blocks. Hence, following the usual notation, we write $x \sigma y$ to denote that $x$ and $y$ are in the same block.

block, $[x]_\sigma$ to denote the block of $\sigma$ that contains the data element $x$ and $D/\sigma$ to denote the set of blocks or equivalence classes of $\sigma$.

**Replacement Policies.** Given a set $D$ of $n$ data items, a cache of size $m$, a data placement scheme $\sigma$, and a sequence $R \in D^N$ of accesses to data items, a replacement policy is a function that decides which block must be evicted from the cache at each time. Formally, a replacement policy is a function $\pi : \{0, 1, 2, \ldots, N\} \rightarrow (D/\sigma)$ that assigns to each time point $i$, the set of blocks that are present in the cache right after the access $R[i]$. Any such policy must satisfy the following:

- $\pi(0) = \emptyset$, i.e. the cache must be empty at the beginning;
- For all $1 \leq i \leq N$, $|\pi(i)| \leq m$, i.e. there are at most $m$ blocks in the cache at each time;
- For all $1 \leq i \leq N$, $|\pi(i) \setminus \pi(i+1)| \leq 2$ and $|\pi(i+1) \setminus \pi(i)| \leq 1$, i.e. at most one block can be added to the cache and at most one block can be evicted at each step;
- For all $1 \leq i \leq N$, $R[i] \in \cup \pi(i)$, i.e. the block containing an access $R[i]$ must be in the cache right after that access.

**Remark 2.1.** Note that the replacement policy only matters when the cache has a size of at least 2. When the cache has unit size, there is always a unique choice for the block that must be evicted.

**Cache Misses.** Given a data placement scheme $\sigma$ and a replacement policy $\pi$ as above, the number of cache misses caused by $\sigma$ and $\pi$ over $R$ is defined as the number of times a new block is loaded into the cache. Formally, $\text{misses}(\sigma, \pi) = \{|i| 1 \leq i \leq N, \pi(i) \setminus \pi(i+1) \neq \emptyset\}$.

**The LRU Policy.** Due to its popularity, we assume throughout this paper that the replacement policy is LRU, i.e. the Least-Recently-Used block is always evicted from the cache. However, most of our results carry over to First-In-First-Out (FIFO) and the Optimal Offline Policy (OOP), as well. Recall that FIFO evicts the oldest block in the cache and OOP evicts the block that is going to be used furthest in the future.

**The Data Packing Optimization Problem.** Consider a memory subsystem that consists of $n$ distinct data elements and a fully-associative cache with a capacity of $m$ blocks and a packing factor of $p$. Given a sequence $R$ of length $N$ of references to data elements, the Data Packing problem asks for a data placement scheme $\sigma$ that minimizes the number of cache misses incurred by the reference sequence $R$, using LRU as the replacement policy. We denote an instance of the Data Packing problem by $I = (n, m, p, R)$.

**Parameters.** In the sequel, we consider the parameters $m$ and $p$ to be small constants and try to find polynomial algorithms in terms of $N$ and $n$.

We now define the concepts of access graph and access hypergraph. Various similar notions have been defined in the past, and are sometimes called affinity graphs or proximity graphs. These hypergraphs will later serve as a basis for reducing the Data Packing problem to a graph problem.

**Access Graph.** Given a sequence $R$ of length $N$ of accesses to data elements from a set $D$ of size $n$, the access graph of $R$ is a simple graph $G_R = (V, E)$ in which $V$ consists of $n$ vertices, each corresponding to one of the data elements in $D$, and there is an edge between two distinct vertices iff their corresponding data elements appear consecutively somewhere in $R$. More formally, $\{u,v\} \in E$ iff $u \neq v$ and there exists an index $i$, such that $\{R[i], R[i+1]\} = \{u,v\}$.

Intuitively, one can think of the graph $G_R$ as the structure on data elements that is respected by the access sequence $R$, in the sense that $R$ can only go from a vertex in $G_R$ to one of its neighbors. Moreover, $G_R$ is the sparsest graph over which $R$ is a (non-simple) path.

**Example 2.1.** Consider the access sequence $R = \langle a, b, c, a, b, d, b, d, e, c, b, f \rangle$. There are 6 data elements in this sequence and its access graph $G_R$ is shown in Figure 2. Note that $R$ is a path on this graph and every edge appears somewhere along $R$, hence no subgraph of $G_R$ has the same property.
We now extend the concept of access graphs to higher order affinity relations between data items, resulting in access hypergraphs.

**Hypergraphs and Ordered Hypergraphs.** A hypergraph \( G = (V, E) \) consists of a set \( V \) of vertices and a multiset \( E \) of hyperedges. Each hyperedge \( e \in E \) is in turn a subset of the vertices of \( G \). An ordered hypergraph \( G = (V, E) \) consists of a set \( V \) of vertices and a set \( E \) of ordered hyperedges. Each ordered hyperedge \( e \in E \) is a sequence of distinct vertices of \( G \), i.e. a hyperedge together with an order on its vertices. Intuitively, hypergraphs are natural extensions of graphs, where each edge can connect more than two vertices. Given a hypergraph \( G \), its primal graph \( G^p \) is a graph on the same set \( V \) of vertices, where two vertices \( u \) and \( v \) are connected by an edge iff there exists a hyperedge \( e \in E \) containing both \( u \) and \( v \). We shall simply refer to hypergraphs and hyperedges as graphs and edges when there is no fear of confusion.

**Access Hypergraph.** Given a natural number \( q \) and an access sequence \( R \) as above, the access hypergraph \( G_R^q = (V, E) \) is a hypergraph defined as follows:

- There are \( n \) vertices in \( V \), each corresponding to one data element;
- For each data access \( R[i] \), there is a corresponding hyperedge \( e_i \) in \( E \). The hyperedge \( e_i \) consists of \( R[i] \) and the \( q - 1 \) distinct data elements that are accessed right before \( R[i] \). If there are less than \( q - 1 \) such elements, \( e_i \) will include all of them. Concretely, \( e_i \) is defined as follows: \( e_i := \{ R[j] \mid j \leq i \land \{ R[j], R[j + 1], \ldots, R[i] \} \leq q \} \).

We call \( q \) the order of the access hypergraph. It is easy to verify that removing repeated edges from the access hypergraph \( G_R^q \) leads to the access graph \( G_R \).

**Example 2.2.** Consider the access sequence \( R =< a, b, c, a, b, d, b, d, e, c, b, f > \). Letting \( q = 3 \), the corresponding access hypergraph \( G_R^3 \) of order 3 consists of the following hyperedges (sometimes there are multiple copies of the same hyperedge, as shown below). We consider these to be distinct hyperedges: \( \{ a \}, \{ a, b \}, \{ a, b, c \} \times 4, \{ a, b, d \} \times 3, \{ b, d, e \}, \{ c, d, e \}, \{ b, c, e \}, \{ b, c, f \} \). Figure 3 shows the segments of the sequence that correspond to edges in \( G_R^3 \).

**Ordered Access Hypergraphs.** Given an access sequence \( R \) as above, the ordered access hypergraph \( \hat{G}_R^q \) is defined similarly to \( G_R^q \), except that each hyperedge is ordered in the natural way, i.e. in the order of appearance of its corresponding data elements in \( R \). Formally, for every access \( R[i] \), there is a corresponding ordered hyperedge \( e_i \) in \( \hat{G}_R^q \). The ordered hyperedge \( e_i \) is a sequence \( < v_1, v_2, \ldots, v_l > \) of vertices of \( \hat{G}_R^q \) such that \( v_l = R[i] \). \( v_{l-1} \) is the first distinct data element accessed before \( R[i] \), \( v_{l-2} \) is the second distinct element, etc. Moreover, \( l \) is the maximum between \( q \) and the number of distinct elements accessed up until \( R[i] \).
**Example 2.3.** Consider the access sequence $R = \langle a, b, c, a, b, d, e, c, b, f \rangle$. The access hypergraph $G^3_R$ was shown in Example 2.2. We now construct the ordered hyperedges of $\tilde{G}^3_R$. Intuitively, we start from any access $R[i]$ in $R$ and go back until we see 3 different data elements. These data elements will form the ordered hyperedge $e_i$ corresponding to $R[i]$. This is illustrated in Figure 4. Note that the elements in an ordered hyperedge $e_i$ are ordered by their last access time before or at $R[i]$, e.g. see the hyperedge $\langle a, d, b \rangle$ in Figure 4.

### 2.2 Tree Decompositions and Treewidth

In parameterized complexity, treewidth is one of the most widely-used parameters for graph problems. It is a measure of how "tree-like" a given graph is. In this section, we provide a quick overview of tree decompositions and treewidth. For an in-depth treatment see [Cygan et al. 2015].

**Tree Decomposition.** Given a (hyper)graph $G = (V, E)$, a tree decomposition of $G$ is a pair $(T, \{X_t \mid t \in T\})$ where $T$ is a tree and each node $t$ of $T$ is associated with a subset $X_t \subseteq V$ of vertices of $G$, such that the following conditions are met:

(i) Every vertex appears in some $X_t$, i.e. $\cup_{t \in T} X_t = V$;
(ii) Every (hyper)edge appears in some $X_t$, i.e. $\forall e \in E \exists X_t \ e \subseteq X_t$;
(iii) For every vertex $v \in V$, the set $T_v = \{t \in T \mid v \in X_t\}$ of all nodes of the tree $T$ that contain $v$ in their corresponding $X_t$, forms a connected subtree of $T$.

It is evident from the definition that $(T, \{X_t\})$ is a tree decomposition of a hypergraph $G$ if and only if it is a tree decomposition of its primal graph $G^P$. To avoid confusion, we reserve the word "vertex" for vertices of $G$ and use the word "node" for vertices of $T$. Moreover, we call each $X_t$ a "bag".

**Treewidth.** The width of a tree decomposition $(T, \{X_t\})$ is the size of its largest bag minus 1, i.e. $\min_{t \in T} |X_t| - 1$. The treewidth of a graph $G$ is the smallest width among all tree decompositions of $G$ and is denoted $tw(G)$.

**Example 2.4.** Figure 5 shows the graph $G_R$ (as in Figure 2) and a tree decomposition of $G_R$. This tree decomposition has a width of 2 and is an optimal tree decomposition. Hence, the treewidth of $G_R$ is 2.

![Fig. 5. A graph $G_R$ (left) and one of its optimal tree decompositions $(T, \{X_t\})$ (right).](image)

To simplify the algorithms that exploit tree decompositions, we now define the notions of labeling and nice tree decomposition.
**Nice Tree Decompositions.** A nice tree decomposition [Cygan et al. 2015] of a (hyper)graph $G$ is a tree decomposition $(T, \{X_t\})$ in which a specific node is designated as the root and every node $t \in T$ is “labeled” by a subgraph $G_t$ of $G$, such that the following rules are obeyed:

1. If $t$ is a leaf in $T$, then $X_t = \emptyset$ and $G_t = (\emptyset, \emptyset)$.
2. Otherwise, $t$ satisfies one of the following cases:
   - **Join Node.** The node $t$ has two children, $t_1$ and $t_2$, $X_t = X_{t_1} = X_{t_2}$ and $G_t = G_{t_1} \cup G_{t_2}$.
   - **Introduce Vertex Node.** The node $t$ has a single child $t_1$ and $X_t = X_{t_1} \cup \{v\}$ for some vertex $v \notin X_{t_1}$. In this case, we say that $t$ introduces $v$. Moreover, $G_t = G_{t_1} \cup \{v\}$, i.e. $G_t$ is defined as the graph resulting from adding $v$ as an isolated vertex to $G_{t_1}$.
   - **Introduce Edge Node.** Similar to the previous case, $t$ has a single child $t_1$. This time, $X_t = X_{t_1}$, but $G_t$ is defined as the graph resulting from adding a new edge $e$ to $G_{t_1}$. All vertices of $e$ must be present in $X_t$. We say that $t$ introduces $e$.
   - **Forget Vertex Node.** The node $t$ has a single child $t_1$ and $X_t = X_{t_1} \setminus \{v\}$ for some vertex $v \in X_{t_1}$. We say that $t$ forgets $v$. Moreover, $G_t = G_{t_1}$.
3. Each edge is introduced exactly once.

Intuitively, the label graph $G_t$ is the subgraph of $G$ consisting of all the vertices and edges that are introduced in the subtree of $T$ rooted at $t$.

**Remark 2.2.** Note that in our (ordered) hypergraphs in this paper, we might have multiple copies of the same (ordered) hyperedge. We treat these as distinct edges and require that each of them be introduced separately in nice tree decompositions.

**Remark 2.3.** The notion of label graphs $G_t$ is solely defined for theoretical purposes and used in our proofs of correctness. In practice, our implementation avoids the overhead of constructing $G_t$’s.

**Example 2.5.** Figure 6 shows a nice tree decomposition of the graph $G$ of Figure 2. In each node $t$ of the tree, its label subgraph $G_t$ is illustrated and the vertices of the bag $X_t$ are shown in red. Intuitively, a nice tree decomposition constructs the graph in small increments and the bag $X_t$ contains the vertices that can participate in the incremental change.

![Fig. 6. A nice tree decomposition of the graph in Figure 2. The leftmost node is the root. The graph $G_t$ is illustrated in each node $t$. The vertices of the bags $X_t$ are shown in red.](image)

**2.3 Existing Results**

We now formally present known results regarding Data Packing and Tree Decompositions that will be used in the sequel.

**The Hardness of Data Packing.** Note that we are considering the problem of minimizing cache misses, not that of maximizing cache hits. While the two problems are equivalent in terms of exact algorithms, approximating the minimal number of cache misses is much harder than approximating the maximal number of cache hits. The latter problem admits a polynomial-time constant-factor

In contrast, the following theorem shows that the former problem is hard to even approximate.

**Theorem 2.1 ([Lavaee 2016]).** Assuming either LRU, FIFO or OOP as the replacement policy, we have the following hardness results:

- For any \( m \) and any \( p \geq 3 \), Data Packing is \( \text{NP-hard} \).
- Unless \( P=\text{NP} \), for any \( m \geq 5 \), \( p \geq 2 \) and any constant \( \epsilon > 0 \), there is no polynomial algorithm that can approximate the Data Packing problem within a factor of \( O(N^{1-\epsilon}) \).

We now turn to tree decompositions. What makes tree decompositions a very useful tool is the fact that one can perform bottom-up dynamic programming on them in a manner similar to trees.

**Separators.** Given a (hyper)graph \( G = (V, E) \), and two sets of vertices \( A, B \subseteq V \), we say that the pair \( (A, B) \) is a separation of \( G \) if \( A \cup B = V \) and no (hyper)edge in \( E \) contains vertices of both \( A \setminus B \) and \( B \setminus A \). We call \( A \cap B \) the separator corresponding to the separation \( (A, B) \) and the order of the separation \( (A, B) \) is the size of its separator \( |A \cap B| \).

**Lemma 2.1 (Separation Lemma, [Bodlaender 1988; Cygan et al. 2015]).** Let \( (T, \{X_t\}) \) be a tree decomposition of \( G \), where \( G \) is a graph or a hypergraph, and let \( \{a, b\} \) be an edge of \( T \). By removing the edge \( \{a, b\} \), \( T \) breaks into two connected components \( T_a \) and \( T_b \), respectively containing \( a \) and \( b \). Let \( A = \bigcup_{t \in T_a} X_t \) and \( B = \bigcup_{t \in T_b} X_t \). Then \( (A, B) \) is a separation of \( G \) with separator \( X_a \cap X_b \).

In our algorithms in the rest of this paper, we assume that whenever a (hyper)graph \( G \) of constant treewidth appears as an input to an algorithm, the input also contains an optimal nice partitioning of the set \( V \), a partitioning such that each partition set has a size of at most \( p \). In other

---

**3 DATA PACKING ON CONSTANT-TREewidth ACCESS GRAPHS**

We now consider the problem of Data Packing when parameterized by the treewidth of the underlying access graph. In Section 3.1, we provide a linear-time algorithm when \( m = 1 \) and the access graph has constant treewidth. Note that this problem is \( \text{NP-hard} \) for general access graphs, as demonstrated by Theorem 2.1. Then, in Section 3.2 we show that for \( m \geq 2 \) the problem remains \( \text{NP-hard} \) and hard-to-approximate even when the access graph is a tree, i.e. has treewidth 1.

**3.1 Algorithm for \( m = 1 \) and Constant-treewidth Access Graph**

We are given a Data Packing instance \( I = (n, 1, p, R) \), its access graph \( G_R \) and a nice tree decomposition \( (T, X_t) \) of the access graph with width \( k \) and \( O(n \cdot k) \) nodes. We first reduce the problem of Data Packing to a graph problem over \( G_R \) and then provide a linear-time fixed-parameter algorithm for solving the graph problem. We start by defining the minimum-weight \( p \)-partitioning problem.

**\( p \)-partitionings.** Given an integer \( p > 0 \) and a graph \( G = (V, E) \), a \( p \)-partitioning of \( G \) is a partitioning \( \psi \) of the set \( V \) of vertices such that each partition set has a size of at most \( p \). In other
words, a $p$-partitioning of $G$ is a data placement scheme where the vertices of $G$ are the data elements and $p$ is the packing factor.

**Cross Edges.** Given a $p$-partitioning $\psi$ of the graph $G = (V, E)$, an edge $e = \{u, v\} \in E$ is called a cross edge if its two endpoints are in different partition sets, i.e. if $[u]_\psi \neq [v]_\psi$.

**Minimum-weight $p$-partitioning.** Given a simple graph $G = (V, E)$, a weight function $w : E \rightarrow \mathbb{N}$ and a positive integer $p$, the minimum-weight $p$-partitioning problem asks for a $p$-partitioning of $G$ in which the total weight of cross edges is minimized.

**Reduction of Data Packing to Minimum-weight $p$-partitioning.** We now reduce the Data Packing problem to minimum-weight $p$-partitioning. Given an instance $I = (n, 1, p, R)$ of Data Packing, we consider the access graph $G_R = (V, E)$ and define the weight function $w_R : E \rightarrow \mathbb{N}$ as $w_R(\{u, v\}) := |\{i \mid \{R[i], R[i+1]\} = \{u, v\}\}|$. Informally, the weight of an edge is the number of times its two endpoints have appeared consecutively in $R$. The reduction is now complete.

**Lemma 3.1.** The optimal number of cache misses in a Data Packing instance $I = (n, 1, p, R)$ is $1$ plus the total weight of cross edges in a minimum-weight $p$-partitioning of $G_R$ with weight function $w_R$.

**Proof.** Every $p$-partitioning $\psi$ of $G_R$ is a data placement scheme for $I$ and vice versa. Given that $m = 1$, the replacement policy does not matter (Remark 2.1) and a cache miss occurs each time $R$ accesses a new block. If we consider $R$ as a path on $G_R$, a cache miss occurs at the very beginning and then each time this path goes from one equivalence class of $\psi$ to another. Therefore, the number of cache misses of $\psi$ is $1$ plus the total weight of cross edges in $\psi$. \hfill \Box

**Example 3.1.** Consider the access sequence $R = \langle a, b, c, a, b, d, b, d, c, b, f \rangle$ of Example 2.1 and the Data Packing instance $I = (6, 1, 2, R)$, i.e. each block can store up to 2 data elements. Figure 7 shows the graph $G_R$ in which every edge is weighted by the number of times it is traversed in $R$. An optimal 2-partitioning of $G_R$ is shown in which vertices of the same color are in the same partition. The total weight of cross edges in this partitioning is 7. The corresponding data placement scheme is $\{\{a, c\}, \{b, d\}, \{e\}, \{f\}\}$ which leads to 8 cache misses on $R$. The cache misses are underlined.

We will provide an algorithm for solving the minimum-weight $p$-partitioning problem on a graph $G$ using an optimal nice tree decomposition of $G$. Our algorithm employs a bottom-up dynamic programming technique. We first need several basic concepts to define the algorithm.

**States over a Set of Vertices.** Given a graph $G = (V, E)$, a natural number $p$ and a subset $A \subseteq V$ of vertices, a state over $A$ is a pair $s = (\varphi, sz)$ such that (i) $\varphi$ is a partitioning of $A$ in which every equivalence class has a size of at most $p$, and (ii) $sz$ is a size enlargement function $sz : A/\varphi \rightarrow \{0, \ldots, p - 1\}$ that maps each equivalence class $[v]_{\varphi}$ to a number which is at most $p - |[v]_{\varphi}|$. Intuitively, the idea is to take $A$ to be one of the bags in the tree decomposition and later extend a state over $A$ to a $p$-partitioning of $G$ by adding the vertices in $V \setminus A$. So, a state over $A$ partitions the vertices of $A$ into sets of size at most $p$ and for each partition $[v]_{\varphi}$ fixes the exact number $sz([v]_{\varphi})$ of vertices from $V \setminus A$ that should be added to $[v]_{\varphi}$. We denote the set of all states over $A$ by $S_{A,p}$ or simply $S_A$ when $p$ is clear from the context.
Efficient Parameterized Algorithms for Data Packing

Fig. 8. All possible states over \( A = \{a, b, c\} \) with \( p = 2 \)

Fig. 9. Two compatible states over \( A = \{a, b, c\} \) and \( A' = \{d, e, f\} \)

**Realization.** We say that a \( p \)-partitioning \( \psi \) realizes the state \( s = (\varphi, sz) \) over \( A \), if (i) the restriction of \( \psi \) to \( A \) is equal to \( \varphi \), i.e. \( \psi|_A = \varphi \) and (ii) for all vertices \( v \in A \), \( sz([v]_{\psi}) = |[v]_{\psi}| - |[v]_{\varphi}| \). Intuitively, \( \psi \) realizes \( s \) if (i) \( \psi \) partitions the vertices in \( A \) in the same manner as \( \varphi \) and (ii) if a partition \( [v]_{\psi} \) of \( \psi \) intersects \( A \), then \( [v]_{\psi} \) contains as many vertices from outside of \( A \) as fixed by \( sz \).

**Example 3.2.** Figure 8 shows all 14 possible states over the set \( A = \{a, b, c\} \) of vertices with \( p = 2 \). In each case, each row denotes one partition set and hence the order of rows and the order of squares in a row does not matter. Empty squares correspond to the possibility of extension of the set, as defined by \( sz \). The optimal 2-partitioning \( \psi \) presented in Figure 7 realizes the highlighted state in Figure 8, because \( \psi \) puts \( a \) and \( c \) in the same partition and puts \( b \) in a partition of size 2, whose other member, \( d \), comes from outside the set \( \{a, b, c\} \).

**Compatibility.** We say that two states \( s \) and \( s' \), respectively over the sets \( A \) and \( A' \), are compatible if there exists a \( p \)-partitioning that realizes both of them. We write \( s \Leftrightarrow s' \) to show compatibility.

**Example 3.3.** Intuitively, two states are compatible if they can fit into each other. Figure 9 shows the states realized by the 2-partitioning of Figure 7 above over the sets \( A = \{a, b, c\} \) and \( A' = \{d, e, f\} \) and how they can be fitted together to create the entire 2-partitioning.

**Algorithm 1.** We are now ready to describe our algorithm in detail. Given a graph \( G \), a weight function \( w \) and an optimal nice tree decomposition \( T \) of \( G \), our algorithm performs a bottom-up dynamic programming on \( T \). This is broken into three steps.

**Step 0: Initialization.** We define several variables at each node of our tree \( T \). These variables are meant to be computed in a bottom-up manner. Concretely, for every \( t \in T \) and every state \( s \) over the bag \( X_t \), we define a variable \( dp[t, s] \) and initialize it to \( +\infty \).

**Invariant.** Formally, our algorithm satisfies the following invariant for every \( dp \) variable right after the end of its computation:

\[ dp[t, s] = \text{The minimum total weight of cross edges over all } p \text{-partitionings of } G_t \text{ that realize } s. \]

Intuitively, we are considering the states over the bag \( X_t \) and extending them by adding vertices that were introduced in the subtree of \( t \) in \( T \).
Step 1: Computation of $dp$. The algorithm starts from the bottom of the tree $T$ and computes the $dp$ variables bottom up, i.e. with an order such that for every node $t \in T$ the $dp$ variables at its children are computed before the $dp$ variables of $t$. For every node $t \in T$ and state $s = (\phi, sz) \in S_{X_t}$, we show how $dp[t, s]$ is computed based on the type of the node $t$:

1. (1.1) if $t$ is a Leaf: $dp[t, s] = 0$;
2. (1.2) if $t$ is a Join node with children $t_1$ and $t_2$:
   $$dp[t, s] = \min_{s_{z1} + sz_2 = sz} dp[t_1, (\phi, sz_1)] + dp[t_2, (\phi, sz_2)];$$
   Note that the summation and equality above are pointwise.
3. (1.3) if $t$ is an Introduce Vertex node, introducing $v$, with a single child $t_1$:
   $$dp[t, s] = dp[t_1, (\phi|_{X_{t_1}}, sz|_{X_{t_1}})];$$
4. (1.4) if $t$ is an Introduce Edge node, introducing $e$, with a single child $t_1$:
   $$dp[t, s] = dp[t_1, s] + w(e, \phi),$$
   where $w(e, \phi)$ is equal to $w(e)$ if $e$ is a cross edge in $\phi$ and zero otherwise;
5. (1.5) if $t$ is a Forget Vertex node, forgetting $v$, with a single child $t_1$:
   $$dp[t, s] = \min_{s' \in S_{X_{t_1}}} dp[t_1, s].$$

Recall that $\leftrightarrow$ denotes compatibility.

Step 2: Computing the Output. The algorithm computes the output, i.e. the optimal weight of a $p$-partitioning, using the values stored at $dp$ variables. If $r$ is the root node of $T$, then the algorithm outputs the following value: $\min_{s \in S_r} dp[r, s]$.

This concludes Algorithm 1. We now prove the correctness of our algorithm.

**Lemma 3.2.** Algorithm 1 correctly computes the total weight of cross edges in a minimum-weight $p$-partitioning.

**Proof.** We prove this lemma in two steps. First, we show that the invariant defined above holds after computing $dp[t, s]$ assuming that it was satisfied for all $dp$ variables in the children of $t$ (Correctness of Step 1). Then, assuming that the invariant holds for $dp$ variables at the root, we show that the output is the total weight of an optimal $p$-partitioning (Correctness of Step 2).

Intuitively, the invariant says that if we only consider the graph $G_t$, i.e. the part of $G$ that was introduced in the subtree of $T$ rooted at $t$, and those $p$-partitionings of $G_t$ that realize the state $s$, then $dp[t, s]$ holds the minimum total weight of cross edges among these $p$-partitionings.

**Correctness of Step 1.** As in the algorithm, we break this part into several cases:

1. (1.1) Computations at Leaves. The node $t$ is a leaf in $T$, hence $G_t$ is the empty graph and $X_t$ is the empty set. Therefore, $S_{X_t}$ contains a single trivial state $s_0$ and we have $dp[t, s_0] = 0$ because the total weight of cross edges in an empty graph is zero.
2. (1.2) Computations at Join Nodes. The node $t$ is a join node with children $t_1$ and $t_2$. We want to compute $dp[t, s]$ where $s = (\phi, sz)$, such that the invariant is satisfied. Therefore, we only consider those $p$-partitionings that realize $s$. Given that $X_t = X_{t_1} = X_{t_2}, \phi$ imposes itself on both $X_{t_1}$ and $X_{t_2}$. However, each partition in $\phi$ must be extended by a number of vertices as defined by $sz$. These vertices must come from either $G_{t_1}$ or $G_{t_2}$ and must not already be present in $X_t$. According to the separation lemma (Lemma 2.1), the only vertices that are in both $G_{t_1}$ and $G_{t_2}$ are precisely those of $X_t$. Hence, each new vertex comes either from $G_{t_1}$ or $G_{t_2}$ but not from both. Therefore, we should minimize our total cross edge weights wrt $dp$ variables of the form $dp[t_1, (\phi, sz_1)]$ and $dp[t_2, (\phi, sz_2)]$ where $sz_1 + sz_2 \equiv sz$. The function $sz_1$
defines the number of vertices that should be added from $G_{t_1} - X_t$ to each partition of $\varphi$ and $sz_2$ does the same for $G_{t_2} - X_t$. Formally, if we let $w(\varphi)$ be the total weight of cross edges caused by $\varphi$ in $G_{t_1} \cap G_{t_2} = G_{t_1}[X_t] \cap G_{t_2}[X_t]$, then we should let:

$$dp[t, s] = dp[t, (\varphi, sz)] = \min_{sz_1 + sz_2 = sz} dp[t_1, (\varphi, sz_1)] + dp[t_2, (\varphi, sz_2)] - w(\varphi).$$

The reason we are subtracting $w(\varphi)$ at the end is that the weights of its corresponding edges are taken into account twice, i.e. once in each of $dp[t_1, (\varphi, sz_1)]$ and $dp[t_2, (\varphi, sz_2)]$.

We now show it is always the case that $w(\varphi) = 0$. If an edge contributes to $w(\varphi)$, then it must be present in both $G_{t_1}$ and $G_{t_2}$. However, by property (3) of a nice tree-decomposition, each edge is introduced exactly once. Hence, $G_{t_1}$ and $G_{t_2}$ do not share any edges and $w(\varphi) = 0$. Therefore, by setting $dp[t, s] = dp[t, (\varphi, sz)] = \min_{sz_1 + sz_2 = sz} dp[t_1, (\varphi, sz_1)] + dp[t_2, (\varphi, sz_2)]$, we satisfy the invariant.

![Fig. 10. In a join node $t$, $G_{t_1}$ and $G_{t_2}$ do not share any edges and their shared vertices are in $X_t$.](image)

(1.3) *Computations at Introduce Vertex Nodes.* The node $t$ is an introduce vertex node. So, it has a single child $t_1$ and $X_t = X_{t_1} \cup \{v\}$ for some $v \notin X_{t_1}$. We know that the vertex $v$ cannot possibly appear in $G_{t_1}$ because every vertex appears in a connected subtree of $T$ and $v \notin X_{t_2}$. Hence, $G_t$ is obtained by adding $v$ as an isolated vertex to $G_{t_1}$. Again, we want to compute $dp[t, s]$ and should hence only consider the $p$-partitionings that realize $s$. Given that $X_{t_1} \subset X_t$, $s$ imposes a unique compatible state on $X_{t_1}$. Moreover, $G_t$ has no new edges in comparison with $G_{t_1}$, so the total weight of cross edges should only be computed in $G_{t_1}$. Hence, we let

$$dp[t, s] = dp[t, (\varphi, sz)] = dp[t_1, (\varphi|_{X_{t_1}}, sz|_{X_{t_1}})].$$

Intuitively, this is equivalent to removing $v$ from its partition and then computing the $dp$ in $t_1$.

![Fig. 11. In an introduce vertex node $t$, the introduced vertex is isolated and there are no new edges.](image)

(1.4) *Computations at Introduce Edge Nodes.* The node $t$ has a single child $t_1$ and $X_t = X_{t_1}$. Moreover, the only difference between $G_t$ and $G_{t_1}$ is in a single edge $e$. When computing $dp[t, s]$, the state $s$ forces itself on $X_{t_1} = X_t$. Hence we should let $dp[t, s] = dp[t_1, s] + w(e, s)$, where $w(e, s)$ is the contribution of the edge $e$ to the total weight of cross edges in $s$. It is zero if the two sides of $e$ are put in the same partition set by $s$ and is equal to $w(e)$ otherwise.
Computations at Forget Vertex Nodes. In this case the node \( t \) has a single child \( t_1 \) and \( X_t = X_{t_1} \setminus \{v\} \) for some \( v \in X_{t_1} \). However, \( G_t = G_{t_1} \). Hence, when computing \( \text{dp}[t, s] \), it is sufficient to take the minimum among the values of \( \text{dp} \) variables of all states \( s' \) over \( X_{t_1} \) that are compatible with \( s \). More precisely, we let

\[
\text{dp}[t, s] = \min_{s' \in S_{X_{t_1}} \land s' \leftrightarrow s} \text{dp}[t_1, s'].
\]

These compatible states are obtained by either (i) putting the vertex \( v \) in its own partition set, or (ii) adding \( v \) to another partition set in \( X_{t_1} \).

Correctness of Step 2. Given that \( r \) is the root node of \( T \), we have \( G_r = G \). Since every \( p \)-partitioning of \( G \) realizes some state over \( X_r \), it follows that the optimal weight of a \( p \)-partitioning is \( \min_{s \in S_{X_r}} \text{dp}[r, s] \). This concludes the proof.
The algorithm computes values of dp variables for all nodes of the tree decomposition which are at most $O(n \cdot k)$. We obtain upper-bounds for the runtime of our algorithm on each type of node:

- **Leaves.** There is a single state at each leaf and its dp is zero. Hence we spend $O(1)$ at each leaf.
- **join Nodes.** At a join node $t$, there are at most $C_k$ states and for each state $s = (\varphi, s_2)$ we have to look into the states corresponding to every possible size enlargement function $sz_1 \leq sz_2$. There are at most $p^k$ such functions. Creating each corresponding state takes $O(k)$. Hence, we spend $O(k \cdot C_k \cdot p^k)$ at each join node.
- **Introduce Vertex Nodes.** At a node $t$, there are $C_k$ states and we spend $O(k)$ computing the unique corresponding state over $X_t$. Thus, each introduce vertex node takes $O(k \cdot C_k)$.
- **Introduce Edge Nodes.** This case is similar to the previous one and takes $O(k \cdot C_k)$.
- **Forget Vertex Nodes.** At a node $t$, there are $C_k$ states and for each of them we have to look into all its compatible states over $X_t$. Note that such compatible states can be obtained either by putting the vertex $v$ in its own partition set, which can have any size between 1 and $p$, or by adding it to the partition set of another vertex in $X_t$. Hence, there are at most $p + k$ such states and the total processing time of a forget vertex node is $O(k \cdot (p + k) \cdot C_k)$.

Note that the runtime for join nodes dominates the rest. Given that there are $O(n \cdot k)$ nodes in total, the whole computation takes $O(n \cdot k^2 \cdot C_k \cdot p^k)$ time. Finally, the algorithm spends $O(C_k)$ time computing the final result using the dp values at the root. □

**Remark 3.2.** By exploiting treewidth, we provided a linear-time algorithm for finding the exact solution to the Data Packing problem when $m = 1$. Note that in the general case, i.e. without considering parameterization by treewidth, this problem is NP-hard as mentioned in Theorem 2.1.

**Remark 3.3.** We assumed LRU as the replacement policy. However, given that the replacement policy does not matter when the cache has unit size (Remark 2.1), our algorithm is applicable to any replacement policy, including FIFO and OOP.

### 3.2 Hardness of Data Packing on Trees

In this section, we provide a reduction from the general problem of Data Packing to the special case where the access graph is a tree, i.e. has treewidth 1. This reduction leads to hardness results that enhance those of [Lavae 2016] by showing that the problem remains hard even on trees. This indicates that although considering constant treewidth access graphs led to efficient algorithms for the case of $m = 1$, constant treewidth access graphs alone are not sufficient for $m \geq 2$.

**Theorem 3.2 (Hardness of Data Packing on Trees).** Given a Data Packing instance $I = (n, m, p, R)$, we have the following hardness results:

- Hardness of the Exact Problem. For any $m \geq 2$ and any $p \geq 3$, Data Packing is NP-hard even if the underlying access graph $G_R$ is a tree.
- Hardness of Approximation. Unless $P=NP$, for any $m \geq 6, p \geq 2$ and any constant $\epsilon > 0$, there is no polynomial approximation algorithm for the Data Packing problem with an approximation factor of $O(N^{1-\epsilon})$ even if the access graph $G_R$ is a tree.

**Proof.** We provide a linear-time reduction that transforms a Data Packing instance $I = (n, m, p, R)$ to another instance $I' = (n + (m + 1)p, m + 1, p, R')$ such that the access graph $G_{R'}$ is a tree. Both hardness results can then be obtained by applying this reduction to the hardness results of Section 2.1. Given $I$, we introduce $(m + 1)p$ new data elements $d_1, d_2, \ldots, d_{(m+1)p}$. Let $X$ be the sequence $d_1, d_2, \ldots, d_{(m+1)p}, d_{(m+1)p-1}, \ldots, d_1$. We form the sequence $R'$ as follows:

$$d_1, R[1], d_1, R[2], d_1, \ldots, d_1, R[N], d_1, X, X, \ldots, X,$$

$2N+m+2$ times
i.e. we take $R$ and add $d_1$ at its beginning, end and between every two elements of it, then we concatenate the result with $2N + m + 2$ copies of $X$. We let $I' = (n + (m + 1)p, m + 1, p, R')$. Note that the cache in $I'$ has one spot more than the cache of $I$.

By construction, $G'_R$ is a tree, because it consists of a path $d_1, \ldots, d_{(m+1)p}$ and every other vertex of the graph is only connected to $d_1$. We now show that the optimal number of cache misses in $I'$ is exactly $m + 1$ plus the optimal number of cache misses in $I$.

Let $\sigma$ be an optimal data placement scheme for $I'$, then $\sigma$ must necessarily put the $d_i$’s in exactly $m + 1$ blocks, otherwise each $X$ in the sequence $R'$ will lead to at least one cache miss for a total of at least $2N + m + 2$. On the other hand, putting the $d_i$’s in $m + 1$ blocks leads to at most $2N + m + 1$ cache misses, even if all accesses before the $X$’s are missed. In particular, $\sigma$ does not put any element of $R$ in the same block as $d_1$. Therefore, $\sigma$ first leads to a cache miss on the first access to $d_1$, then keeps $d_1$ in the cache forever. Hence, $\sigma$ fills one spot of the cache with the block of $d_1$ and has $m$ spots left for scheduling $R$. Finally, $\sigma$ loads the other $m$ blocks that contain some $d_i$’s but not $d_1$. Hence, the number of cache misses caused by $\sigma$ is 1 (for the first $d_1$) plus the optimal number of misses in $I$ plus $m$ (for the $X$’s).

**Remark 3.4.** As mentioned before, we are considering the LRU replacement policy in this paper. However, the reduction above works for the OOP replacement policy as well. Hence, the hardness results are established for both policies.

## 4 DATA PACKING ON CONSTANT-TREEWIDTH ACCESS HYPERGRAPHS

In this section, we exploit constant treewidth of higher-order access hypergraphs for solving Data Packing. Section 4.1 extends our linear-time algorithm to every $m$, when the access hypergraph of order $q^* := (m - 1)p + 2$ has constant treewidth. As indicated by Theorem 2.1, this problem is hard to even approximate in the general case. In Section 4.2 we argue that $q^*$ is the optimal order for exploiting treewidth in the sense that the problem remains NP-hard even if the access hypergraph of order $q^* − 1$ has constant treewidth. This also leads to a new hardness-of-approximation result.

### 4.1 Algorithm for Constant-treewidth Access Hypergraph

In this section we extend the algorithm of Section 3.1 to any cache size $m$, provided that the hypergraph $G'_R$ is of constant treewidth, where $q^* = (m - 1)p + 2$. Note that Theorem 3.2 implies such an extension cannot be made if we only consider constant-treewidth $G_R$.

**Intuition on Cache Misses.** The main intuition behind our algorithm is the following: given an instance $I = (n, m, p, R)$ and a data placement scheme $\sigma$ for $I$, we can deduce whether an access $R[i]$ leads to a cache miss by looking at only the $(m - 1)p + 1 = q^* - 1$ previous accesses to distinct data elements. We now formalize this intuition.

**Previous Access of a Block.** Consider a Data Packing instance $I = (n, m, p, R)$, a data placement scheme $\sigma$ for $I$ and an access $R[i]$. Let $B := [R[i]]_{\sigma}$ be the block of $\sigma$ containing $R[i]$. We define $prev_{\sigma}(i)$ as the index of the previous access to $B$ or 0 if no such access exists, i.e. $prev_{\sigma}(i) := \max\{j < i | j = 0 \lor [R[j]]_{\sigma} = [R[i]]_{\sigma}\}$.

**Lemma 4.1.** Given a data placement scheme $\sigma$ for $I$, an access $R[i]$ leads to a cache miss if and only if $prev_{\sigma}(i) = 0$ or there are at least $m$ distinct blocks of $\sigma$ whose elements appear in the range $R[prev_{\sigma}(i) + 1] \ldots R[i - 1]$.

**Proof.** We are assuming LRU as the replacement policy and the cache starts empty. If $prev_{\sigma}(i) = 0$, then $R[i]$ is the first access to its block and will definitely lead to a cache miss. We now consider the case where $prev_{\sigma}(i) \neq 0$. Let $j := prev_{\sigma}(i)$ and assume that $B := [R[i]]_{\sigma} = [R[j]]_{\sigma}$ is the block containing $R[i]$ and $R[j]$. By definition, none of the elements $R[j + 1], \ldots, R[i - 1]$ belong to $B$. If
there are at most \( m - 1 \) blocks between \( R[j + 1] \) and \( R[i - 1] \), then \( R[i] \) cannot lead to a cache miss. This is because right after the access \( R[j] \), the block \( B \) is present in the cache and is the most recently used block of the cache. Hence, in order for it to be evicted, at least \( m \) other blocks must be accessed. On the other hand, if there are at least \( m \) blocks between \( R[j + 1] \) and \( R[i - 1] \), then all of these blocks will be loaded into the cache and hence \( B \) will be evicted before the access \( R[i] \) leading to an eventual cache miss on \( R[i] \).

**Corollary 4.1.** Given a data placement scheme \( \sigma \), an access \( R[i] \) and the \( q^* - 1 \) distinct elements that were accessed before \( R[i] \) (or all of the previous distinct elements if there is less than \( q^* - 1 \) of them) in the order of their last access time, one can deduce whether \( R[i] \) leads to a cache miss.

**Proof.** If \( R[\text{prev}_\sigma(i)] \) is one of these previous elements, then we can simply check whether at least \( m \) different blocks appear between \( R[\text{prev}_\sigma(i)] \) and \( R[i] \). Otherwise, either \( R[i] \) is the first access to its block or all the \( q^* - 1 \) elements are appearing between \( R[\text{prev}_\sigma(i)] \) and \( R[i] \). In the first case \( R[i] \) leads to a cache miss. In the second case, by pigeonhole principle, there are at least \( m \) blocks between the two elements \( R[\text{prev}_\sigma(i)] \) and \( R[i] \) and hence there is a cache miss at \( R[i] \).

**Remark 4.1.** Note that the previous access to the block containing \( R[i] \) might be an access to \( R[i] \) itself. Hence, \( R[i] \) might itself appear in the \( q^* - 1 \) distinct elements that were accessed before \( R[i] \).

As in Section 3.1, we are going to reduce Data Packing to a graph problem and then exploit treewidth to obtain a linear-time algorithm. Corollary 4.1 suggests that in order to detect cache misses, one only needs to consider the ordered access hypergraph of order \( q^* = (m - 1)p + 2 \), i.e. \( \hat{G}_R^{q^*} \). However, in order to address the corner case mentioned in Remark 4.1, we define an ordered hypergraph \( G \) by a slight change to the edges of \( \hat{G}_R^{q^*} \) and then reduce Data Packing to a graph problem over \( G \).

**The Ordered Hypergraph** \( G \). We define the ordered hypergraph \( G \) as having the same vertices and edges as the ordered access hypergraph \( \hat{G}_R^{q^*} \), except in the following case:

- Given an access \( R[i] \) to a data element \( d \), let \( R[j] \) be the last access before \( R[i] \) to the same data element \( d \). If there are at most \( q^* \) distinct data elements accessed in the range \( R[j+1] \ldots R[i-1] \), then the edge \( e_j \) corresponding to \( R[j] \), will also contain \( R[j] \) (in its natural position according to the order of vertices in \( e_j \)).

**Example 4.1.** Consider the access sequence \( R = \langle d, c, a, b, c \rangle \) and let \( m = p = 2 \). Hence, we have \( q^* = (m - 1)p + 2 = 4 \). In the graph \( \hat{G}_R^{q^*} \), the edge corresponding to the second \( c \) is \( e_5 = \langle d, a, b, c \rangle \). However, there are less than \( q^* \) distinct data elements appearing between the two accesses to \( c \), i.e. there are only two such elements, namely, \( a \) and \( b \). Hence, in \( G \), the previous access to \( c \) appears in this edge as well. Therefore, in \( G \), the edge \( e_5 \) is of the form \( e_5 = \langle d, c, a, b, c \rangle \).

The intuition behind the way \( G \) is defined comes from Corollary 4.1 and Remark 4.1. The idea is to have the edge \( e_j \) contain all the data necessary to decide whether a cache-miss will happen at the access \( R[i] \). We now formalize this concept.

**Missed Edges.** Given an ordered hyperedge \( e_j \) of \( G \) and a data placement scheme \( \sigma \), we can deduce whether a cache-miss happens at \( R[j] \) using Corollary 4.1, because the edge \( e_j \) contains an ordered list of at least \( (m - 1)p + 1 = q^* - 1 \) distinct data elements that were accessed right before \( R[i] \). We say that an ordered hyperedge \( e_j \) is missed in \( \sigma \), if the corresponding \( R[i] \) is a cache miss.

**Identifying Missed Edges.** Consider the data placement scheme \( \sigma \) as a \( p \)-partitioning of vertices of \( G \). Based on Lemma 4.1 and Corollary 4.1, an ordered hyperedge \( e_j = \langle v_1, \ldots, v_l \rangle \) is missed iff the sequence of vertices \( < v_1, v_{l-1}, \ldots, v_1 > \) in \( G \) visits at least \( m \) distinct partitions before getting...
back to the partition \([v_l]_\sigma\) or if it never comes back. Note that this determination can be done in \(O(m \cdot p)\) and only depends on the \(p\)-partitioning of \(\{v_1, \ldots, v_l\}\).

We now define our graph problem as follows:

**Minimum-miss \(p\)-partitioning.** Given a hypergraph \(G = (V, E)\) with ordered hyperedges, partition \(V\) into sets of size at most \(p\) in a manner that minimizes the number of missed edges.

As a direct result of the previous discussion, we have the following lemma:

**Lemma 4.2.** The optimal number of cache misses in a Data Packing instance \(I = (n, m, p, R)\) is equal to the optimal number of missed edges in a \(p\)-partitioning of \(G\).

**Proof.** Any data placement scheme \(\sigma\) for \(I\) is also a \(p\)-partitioning of \(G\). As shown above, \(\sigma\) misses an edge \(e_i\) in \(G\) iff it causes a cache miss at \(R[i]\) in \(I\). Hence, the number of cache misses caused by \(\sigma\) in \(I\) is equal to the number of missed edges caused by \(\sigma\) in \(G\).

**States over a Set of Vertices.** We define states in the exact same manner as in Section 3.1, i.e. a state over a set \(A\) of vertices is a pair \(s = (\varphi, sz)\) consisting of an equivalence relation \(\varphi\) and a size enlargement function \(sz\). The concepts of realization and compatibility are also defined similarly.

**Algorithm 2.** We now provide a linear-time algorithm for solving the minimum-miss \(p\)-partitioning problem, assuming that the hypergraph \(G\) has constant treewidth. The algorithm is an extension of the one provided in Section 3.1. In the following, we let \((T, \{X_t | t \in T\})\) be an optimal nice tree decomposition of \(G\). Our algorithm performs a bottom-up dynamic programming on \(T\).

**Step 0: Initialization.** We define several variables at each node of the tree \(T\). Concretely, for every node \(t \in T\) and every state \(s\) over \(X_t\), we define a variable \(dp[s, t]\), initially holding a value of \(+\infty\).

**Invariant.** The most different aspect of our algorithm compared to Section 3.1 is the invariant. Formally, we require our algorithm to satisfy the following invariant for every \(dp\) variable right after the end of its computation:

\[
dp[t, s] := \text{The minimum number of missed edges over all } p\text{-partitionings of } G_t \text{ that realize } s.
\]

**Step 1: Computation of \(dp\).** The \(dp\) variables are computed in a bottom-up manner. Given a node \(t \in T\) and a state \(s = (\varphi, sz) \in S_{X_t}\), we show how \(dp[s, t]\) is computed in terms of the \(dp\) variables at the children of \(t\). This computation depends on the type of the node \(t\).

1. (1) if \(t\) is a Leaf: \(dp[t, s] = 0\);
2. (2) if \(t\) is a Join node with children \(t_1\) and \(t_2\):
   \[
dp[t, s] = \min_{sz_1 + sz_2 = sz} \left( dp[t_1, (\varphi, sz_1)] + dp[t_2, (\varphi, sz_2)] \right);
   \]
3. (3) if \(t\) is an Introduce Vertex node, introducing \(v\), with a single child \(t_1\):
   \[
dp[t, s] = dp[t_1, (\varphi|_{X_{t_1}}, sz|_{X_{t_1}})];
   \]
4. (4) if \(t\) is an Introduce Edge node, introducing \(e\), with a single child \(t_1\):
   \[
dp[t, s] = \begin{cases} 1 & \text{missed_edge}(e, \varphi) \\ 0 & \text{otherwise} \end{cases};
   \]
5. (5) if \(t\) is a Forget Vertex node, forgetting \(v\), with a single child \(t_1\):
   \[
dp[t, s] = \min_{s_1 \in S_{X_{t_1}} \land s_1 \leftrightarrow s} dp[t_1, s_1].
   \]

Recall that \(\leftrightarrow\) denotes compatibility of states.
Step 2: Computing the Output. The algorithm computes the output, i.e. the minimum number of missed edges in a \( p \)-partitioning of \( G \), using the values of \( dp \) variables at the root node \( r \) of \( T \). Formally, the output is \( \min_{s \in S_X} \ dp[r, s] \).

This concludes Algorithm 2. While most of the computations are similar to Algorithm 1, the argument for correctness of Algorithm 2 and its runtime are rather different. We first prove the correctness of our approach and then establish its time complexity.

**Lemma 4.3.** Algorithm 2 correctly computes the total number of missed edges in a minimum-miss \( p \)-partitioning.

**Proof.** Our proof heavily depends on the invariant defined above. Intuitively, the invariant says that \( dp[t, s] \) must be filled with the minimum number of edges that are missed in a \( p \)-partitioning realizing \( s \), over the subgraph \( G_t \) of \( G \), which consists of all the vertices and hyperedges that are introduced below \( t \) in \( T \). We prove the lemma in two steps. First, we prove that the invariant is satisfied after computing \( dp[t, s] \), assuming that it were satisfied for all \( dp \) variables in the children of \( t \) (Correctness of Step 1). Then, we prove that assuming the invariant holds for \( dp \) variables at the root node \( r \) of \( T \), the algorithm computes the right output (Correctness of Step 2).

**Correctness of Step 1.** We break the proof into several cases:

1. (1) **Computation at Leaves.** The node \( t \) is a leaf in \( T \). So \( G_t \) is the empty graph and hence there are no missed edges in \( G_t \). Moreover, there is exactly one state over \( X_t \), i.e. the trivial state \( s_0 \). Hence, we should let \( dp[t, s_0] = 0 \).

2. (1.2) **Computation at Join Nodes.** A join node \( t \) has two children \( t_1 \) and \( t_2 \) with \( X_t = X_{t_1} \times X_{t_2} \). When computing the value of \( dp[t, s] \) for a state \( s = (\varphi, sz) \), we only have to consider those states over \( X_{t_1} \) and \( X_{t_2} \) that are compatible with \( s \). Hence, \( X_{t_1} \times X_{t_2} \) gives us the total number of missed edges in \( G_{t_1} \) and \( G_{t_2} \). The function \( sz \) specifies how many new vertices must be added to each partition of \( \varphi \) from \( G_{t_1} \) and \( G_{t_2} \). Note that by the separation lemma (Lemma 2.1), the only vertices that belong to both \( G_{t_1} \) and \( G_{t_2} \) are already included in \( X_t \), hence no new vertex can be in both. Therefore, we have to look into \( dp \) variables of the form \( dp[t_1, (\varphi, sz_1)], dp[t_2, (\varphi, sz_2)] \) where \( sz_1 + sz_2 = sz \). Concretely, we should let:

\[
dp[t, s] = dp[t, (\varphi, sz)] = \min_{sz_1+sz_2=sz} \ dp[t_1, (\varphi, sz_1)] + dp[t_2, (\varphi, sz_2)].
\]

Note that the two graphs \( G_{t_1} \) and \( G_{t_2} \) do not share any edges as argued in Lemma 3.2.

3. (1.3) **Computation at Introduce Vertex Nodes.** In this case, \( t \) is a node, with a single child \( t_1 \), that introduces the vertex \( v \). Then \( v \notin G_{t_1} \) and \( G_t = G_{t_1} \cup \{v\} \), i.e. \( G_t \) is obtained by adding \( v \) to \( G_{t_1} \) as an isolated vertex. Given that \( G_t \) has no new edges in comparison with \( G_{t_1} \), it follows that the missed edges in \( G_t \) are precisely those that were missed in \( G_{t_1} \). Also, \( X_t = X_{t_1} \cup \{v\} \) and so given a state \( s = (\varphi, sz) \) over \( X_t \), there is only one compatible state over \( X_{t_1} \), i.e. \( s_1 = (\varphi|_{X_{t_1}}, sz|_{X_{t_1}}) \). Therefore, we must let \( dp[t, s] = dp[t_1, s_1] \).

4. (1.4) **Computation at Introduce Edge Nodes.** The node \( t \) has one child \( t_1 \), \( X_t = X_{t_1} \) and \( G_t = G_{t_1} \cup \{e\} \), where \( e \) is the introduced hyperedge. Note that, by property (2) of nice tree decompositions, all vertices of \( e \) must appear in \( X_t \). So \( \varphi \) gives us enough information to know whether \( e \) is a missed edge. Also, given that \( X_{t_1} = X_t \), the state \( s \) forces itself on \( X_{t_1} \) and therefore, letting

\[
dp[t, s] = dp[t_1, s] + \begin{cases} 1 & \text{missed}_\text{edge}(e, \varphi) \\ 0 & \text{otherwise} \end{cases}
\]

preserves the invariant.

5. (1.5) **Computation at Forget Vertex Nodes.** This case is handled in the exact same manner as in Section 3.1. Given that \( G_t = G_{t_1} \) and \( X_t \subset X_{t_1} \), the value of \( dp[t, s] \) should be set to the
minimum value of \(dp[t_1, s_1]\) over all states \(s_1\) that are compatible with \(s\). Formally,

\[
dp[t, s] = \min_{s_1 \in S_{C_k}^q \land s_1 \rightarrow s} dp[t_1, s_1].
\]

**Correctness of Step 2.** Let \(r\) be the root node of \(T\), then \(G_r = G\) and every \(p\)-partitioning of \(G\) realizes exactly one state over \(X_r\). Hence, the minimum number of missed edges in the entire graph \(G\) is \(\min_{s \in S_{C_k}^q} dp[r, s]\).

\[\square\]

**Remark 4.2.** Algorithm 2 computes the optimal number of missed edges in a \(p\)-partitioning of \(G\). As is common in dynamic programming approaches, an optimal \(p\)-partitioning itself can be obtained by keeping track of the choices that led to minimum values during the computation of \(dp\) variables.

We conclude this section by establishing the complexity of Algorithm 2.

**Theorem 4.1.** Given a Data Packing instance \(I = (n, m, p, R)\) as input, where \(n\) is the number of distinct data items, \(p\) is the packing factor, \(R\) is the reference sequence with a length of \(N\) and the cache has a capacity of \(m\) blocks, the Data Packing problem, i.e. finding the minimal number of cache misses, can be solved in linear time, i.e. in time \(O(n \cdot k^2 \cdot C_k \cdot p^k + N \cdot C_k \cdot (k + m \cdot p))\), when the underlying access hypergraph \(G_R^q\) has treewidth \(k - 1\).

**Proof.** Creating the ordered hypergraph \(G\) and the reduction from Data Packing to minimum-miss \(p\)-partitioning using Lemma 4.2 takes linear time, i.e. \(O(N \cdot m \cdot p)\). Note that \(G\) is obtained by ordering the vertices of \(G_R^q\) and then adding duplicated vertices to some of the edges, hence \(\text{tw}(G) = \text{tw}(G_R^q)\). As before, the optimal tree decomposition \((T, \{X_i\})\) can be computed in linear time by Lemmas 2.2 and 2.3. Since there are \(N\) hyperedges in \(G\), the tree \(T\) will have \(O(n \cdot k + N)\) nodes, where \(N\) of them are introduce edge nodes and \(O(n \cdot k)\) of them are of the other types.

The times spent at leaves, join nodes, introduce vertex nodes and forget vertex nodes are exactly the same as those established in Theorem 3.1. In an introduce edge node, the algorithm has to compute \(C_k\) different \(dp\) values, each taking time \(O(k + m \cdot p)\) due to the call to the missed_edge subprocedure. Hence, processing each introduce edge node takes \(O(C_k \cdot (k + m \cdot p))\). Therefore, the total time spent on computing \(dp\) values is \(O(n \cdot k^2 \cdot C_k \cdot p^k + N \cdot C_k \cdot (k + m \cdot p))\). Finally, it takes \(O(C_k)\) time to compute the final answer using \(dp\) variables at the root node. \(\square\)

**Remark 4.3.** The runtime above is linear in \(n\) and \(N\), given that \(C_k\) is bounded by a function of \(p\) and \(k\) (see [Chatterjee et al. 2019] for bounds on \(C_k\)). Hence, by exploiting the treewidth of \(G\), we were able to obtain an exact linear-time algorithm for Data Packing. Note that by Theorem 2.1, the general problem, i.e. without parameterization by treewidth, is hard to even approximate.

**Remark 4.4.** Note that every hyperedge of the access hypergraph \(G_R^q\) connects \(q^*\) distinct vertices. Hence, the treewidth of this access hypergraph is at least \(q^* - 1 = (m - 1)p + 1\). Both \(m\) and \(p\) are assumed to be constants. However, this dependency means that the algorithm cannot scale to caches of large size or large packing factor as the involved constants become high.

### 4.2 Hardness of Data Packing on Constant-treewidth Access Hypergraphs

In Section 3.2, we showed that Data Packing is hard even if the access graph \(G_R\) is a tree, i.e. even if \(G_R^q\) has treewidth 1. Section 4.1 provided a linear-time algorithm for Data Packing when \(G_R^q\) has constant treewidth. This naturally leads to the question whether \(q^* = (m - 1)p + 2\) is the optimal order for exploiting treewidth. Note that this is a well-posed problem because for every \(i\), the primal graph of \(G_R^q\) is a subgraph of the primal graph of \(G_R^{i+1}\) and hence \(\text{tw}(G_R^q) \leq \text{tw}(G_R^{i+1})\). Formally,
the question is whether there exists a polynomial algorithm for Data Packing assuming that the hypergraph $G_{R}^{q*−1}$ has constant treewidth. In this section, we show that this problem is NP-hard and hence, unless P=NP, there is no such algorithm and $q^*=\sigma$ is the optimal order. We then show that for a slightly smaller order, i.e. $q^*-4p-1=(m−5)p+1$, the problem becomes hard to approximate.

**Theorem 4.2 (Hardness of Data Packing in Constant Treewidth).** Given a Data Packing instance $I = (n, m, p, R)$, for any cache size $m \geq 2$ and any packing factor $p \geq 3$, Data Packing is NP-hard even if the underlying access hypergraph $G_{R}^{q*−1}$ has constant treewidth.

**Proof.** By Theorem 2.1, we know that Data Packing is NP-hard for any $p \geq 3$ and $m = 1$. We use this problem to obtain our reduction. Formally, for every $m$, we provide a linear-time reduction that transforms the Data Packing instance $I = (n, p, R)$ to a new instance $I' = (n', m, p, R')$ such that the access hypergraph $G_{R'}^{q*−1}$ is of constant treewidth.

Given a positive integer $m$ and an instance $I$ as above, we introduce $(m+1)p$ new data elements $d_1, d_2, \ldots, d_{(m+1)p}$. We then define three sequences $X, Y$ and $Z$ as follows:

$$X := d_1, d_2, \ldots, d_{(m-1)p}, \quad Y := d_{(m-1)p+1}, d_{(m-1)p+2}, \ldots, d_{mp}, \quad Z := d_{mp+1}, d_{mp+2}, \ldots, d_{(m+1)p},$$

and construct the reference sequence $R'$ as:

$$R' := X, R[1], X, R[2], X, \ldots, X, R[N], X, Y, X, Y, \ldots, X, Y, X, Z, X, Z, \ldots, X, Z,$$

i.e. $R'$ is obtained by adding $X$ before every element of $R$ and then appending the result with $a$ copies of $X, Y$ and $b$ copies of $X, Z$. The instance $I$ is then reduced to $I' = (n + (m+1)p, m, p, R')$.

Our goal is to set the right values for $a$ and $b$ in a way that forces any optimal data packing scheme $\sigma$ to put $X$ in exactly $m-1$ blocks. We set $a := N(m-1)p + N + 2m + 1$ and $b := N(m-1)p + N + amp + m + 1$. Using these parameters, every optimal data packing scheme $\sigma$ has to put $X \cup Z$ in exactly $m$ blocks. This is because using more than $m$ blocks for them leads to at least $b$ misses in the last part of the sequence $R'$, while putting them in exactly $m$ blocks can cause a maximum of $N(m-1)p + N + amp + m = b-1$ misses overall, i.e. even if every access up until the end of the last $Y$ leads to a miss. Given that $\sigma$ puts $X \cup Z$ in exactly $m$ blocks, we also infer that $\sigma$ causes at most $m$ misses over the $b$ repetitions of $X, Z$.

We now prove that $\sigma$ has to put $X \cup Y$ in exactly $m$ blocks. The reasoning is similar. If $\sigma$ puts $X \cup Y$ in more than $m$ blocks, it causes at least $a$ cache misses, but if it puts them in exactly $m$ blocks the number of cache misses is at most $N(m-1)p + N + 2m = a - 1$, i.e. even if every access up until $R[N]$ is missed and $\sigma$ misses $m$ times over the repetitions of $X, Z$. Given that $X \cup Z$ and $X \cup Y$ are both put into $m$ blocks, it follows that $\sigma$ puts $X$ in exactly $m-1$ blocks.

We claim that the optimal number of cache misses in $I'$ is $m+1$ plus the optimal number of cache misses in $I$. To see this, we track the behavior of $\sigma$ over the access sequence $R'$. First, the $m-1$ blocks of $X$ are loaded into the cache causing $m-1$ cache misses. These remain in the cache forever because of the way $X$ is repeated in $R'$. Therefore, $\sigma$ has filled $m-1$ spots of the cache with $X$ and has only 1 spot for handling the $R[i]'s$. This leads to exactly as many cache misses as in the optimal solution to $I$. Finally, $\sigma$ causes two more cache misses, one on the first access to $Y$ and the other one on the first access to $Z$. Therefore, the optimal number of cache misses in $I'$ is equal to the optimal number of cache misses in $I$ plus $m+1$. The reduction is now complete.

It remains to show that $G_{R'}^{q*−1} = G_{R'}^{(m-1)p+1}$ has constant treewidth. Figure 14 shows a tree decomposition of this graph with width $mp-1$. Therefore, $tw(G_{R'}^{q*−1}) \leq mp - 1 = O(1)$. 

\[\square\]
We now turn to the hardness of approximation. We provide a reduction that follows the same intuition as in the previous theorem.

**Theorem 4.3 (Hardness of Approximating Data Packing in Constant Treewidth).** Given a Data Packing instance $I = (n, m, p, R)$, for any cache size $m \geq 6$, any packing factor $p \geq 2$ and any constant $\epsilon > 0$, unless P=NP, Data Packing cannot be approximated within a factor of $O(N^{1-\epsilon})$ even if the underlying access hypergraph $G_R^{q-4p-1} = G_R^{(m-5)p+1}$ has constant treewidth. Here, $N$ is the length of the reference sequence $R$.

**Proof.** We know from Theorem 2.1 that for $m = 5$, and $p \geq 2$, it is hard to approximate Data Packing within a factor of $O(N^{1-\epsilon})$. We reduce this problem to Data Packing on a constant-treewidth $G_R^{(m-5)p+1}$. Formally, for every $m \geq 6$, we provide a linear-time reduction from every instance $I = (n, 5, p, R)$ to an instance $I' = (n', m, p, R')$ such that $G_{R'}^{(m-5)p+1}$ has constant treewidth.

The reduction and the argument for its correctness are similar to those of Theorem 4.2. Given an instance $I$ as above, we introduce $(m + 5)p$ new data elements $d_1, d_2, \ldots, d_{(m+5)p}$. We then define the following four sequences in a manner similar to Theorem 4.2:

\[
X := d_1, d_2, \ldots, d_{(m-5)p}; \quad Y := d_{(m-5)p+1}, d_{(m-5)p+2}, \ldots, d_{mp}; \quad Z := d_{mp+1}, d_{mp+2}, \ldots, d_{(m+5)p};
\]

\[
R' := X, R[1], X, R[2], \ldots, X, R[N], X, Y, X, Y, \ldots, X, Y, X, Z, X, Z, \ldots, X, Z,
\]

where $a = N(m-5)p + N + 2m + 1$ and $b = N(m-5)p + N + amp + m + 1$. The reduction is then from $I = (n, 5, p, R)$ to $I' = (n' + (m + 5)p, m, p, R')$. It is straightforward to check that every optimal data placement scheme $\sigma$ has to put each of $X \cup Y$ and $X \cup Z$ in exactly $m$ blocks. Hence, it has to put $X$ in exactly $m - 5$ blocks. Therefore, the optimal number of cache misses in $I'$ is $m - 5$ (for loading the first $X$) plus the optimal number of cache misses in $I$ plus $10$ (5 misses for loading the first $Y$ and 5 for the first $Z$). Finally, the same tree decomposition as in Figure 14, shows that $tw(G_{R'}^{(m-5)p+1}) \leq mp - 1 = O(1)$. \hfill $\Box$

## 5 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

**Implementation and Machine.** We implemented our approach (i.e. Algorithms 1 and 2) in C++. We used a Java library called LibTW [van Dijk et al. 2006] to obtain the tree decompositions. All results are obtained using an Intel Xeon E5-1650v3 3.5GHz processor running 64-bit Debian 8.

**Benchmarks.** We used a variety of classical algorithms to generate the access sequences $R$ for the Data Packing problem. For each algorithm, we generated random inputs of various sizes, which in turn led to random access sequences of varying lengths. We included algorithms from the following categories in our benchmark set: (i) linear algebra, (ii) sorting, (iii) dynamic programming, (iv) recursion, (v) string matching, (vi) computational geometry, (vii) algorithms on trees and (viii) algorithms on sorted arrays. For a complete list of the classical algorithms we used to generate benchmarks, see [Chatterjee et al. 2019]. We implemented these classical algorithms in C++ and...
Experimental Results. Our experimental results over all instances are illustrated in Figure 16. Each row of Figure 16 shows a comparison between our algorithm and a number of heuristics. The $x$-axis denotes the optimal number of cache misses and the $y$-axis the number of cache misses incurred by the heuristic algorithm. Therefore, our algorithms’ results correspond to the $y = x$ line, and, as expected, all heuristic-based results are above or on this line, i.e., they lead to more cache misses than optimal. To give a better view of the results, each row starts with a full plot of all the instances (on the left) and then zooms into the areas with a high density of points, which correspond to the instances that led to relatively few cache misses. We found that in total, our algorithm reduces the number of cache misses by between 15% (over Sampling) to 31% (over $k$-Distance). We also found that our algorithm is effective on every category of benchmarks (see Figure 17).
Efficient Parameterized Algorithms for Data Packing

6 DISCUSSION AND CONCLUSION

We studied if the treewidth property of access hypergraphs can be exploited for efficient algorithms for Data Packing. We showed a fine-grained dichotomy, i.e., we established an optimal value $q^*$ of the order of the access hypergraphs, such that (a) if the access hypergraph of order $q^*$ has constant treewidth, then Data Packing admits a linear-time algorithm; and (b) even if the access hypergraph of order $q^* - 1$ has constant treewidth, in general, the Data Packing problem remains NP-hard. Moreover, we showed a hardness-of-approximation result that holds even if the access hypergraph of order $q^* - 4p - 1$ has constant treewidth. These results are summarized in Figure 1 on Page 5.

Note that while in our algorithms, we focused on LRU as the replacement policy, our results also extend to FIFO and OOP. In particular our first linear-time algorithm (Algorithm 1, Theorem 3.1) applies to all replacement policies, and our second linear-time algorithm (Algorithm 2, Theorem 4.1) applies both to LRU and FIFO, with minor modifications.

While we studied the treewidth property of access hypergraphs from a theoretical perspective, in the experimental results we observed that in practice, many real-world algorithms exhibit the desired property, i.e., the access hypergraphs of the required order have small treewidth. Thus the usefulness of parameterization by the treewidth property is also validated experimentally. In this work, we presented the first positive theoretical results and efficient parameterized algorithms for Data Packing. There are several directions of future work. First, whether other structural properties of access hypergraphs can be exploited for efficient algorithms is an interesting direction. Second, studying the extension of the Data Packing problem for multi-level caches and algorithms for them is another interesting direction. Third, while we provided linear-time algorithms for data packing in constant treewidth, their runtime dependency on constant factors $m, p$ and the treewidth are non-polynomial. Improving this aspect of the runtime is another interesting direction of future work. Finally, studying the Data Packing problem in an online setting where the access sequence is not known in advance is a very challenging direction to pursue.
REFERENCES


