Reordering and Compression for Hypergraph Processing

Yu Liu, Qi Luo, Mengbai Xiao, Dongxiao Yu, *Senior Member, IEEE,* Huashan Chen, and Xiuzhen Cheng, *Fellow, IEEE*

Abstract—Hypergraphs are applicable to various domains such as social contagion, online groups, and protein structures due to their effective modeling of multivariate relationships. However, the increasing size of hypergraphs has led to high computation costs, necessitating efficient acceleration strategies. Existing approaches often require consideration of algorithm-specific issues, making them difficult to directly apply to arbitrary hypergraph processing tasks. In this paper, we propose a compression-array acceleration strategy involving hypergraph reordering to improve memory access efficiency, which can be applied to various hypergraph processing tasks without considering the algorithm itself. We introduce a new metric called closeness to optimize the ordering of vertices and hyperedges in the one-dimensional array representation. Moreover, we present an $\frac{1}{2w}$-approximation algorithm to obtain the optimal ordering of vertices and hyperedges. We also develop an efficient update mechanism for dynamic hypergraphs. Our extensive experiments demonstrate significant improvements in hypergraph processing performance, reduced cache misses, and reduced memory footprint. Furthermore, our method can be integrated into existing hypergraph processing frameworks, such as Hygra, to enhance their performance.

Index Terms—Graph analytics; hypergraph; locality; reordering; optimization.

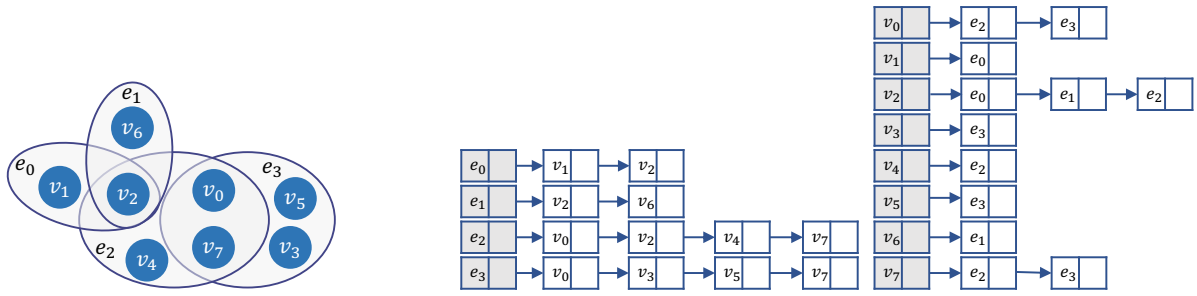
1 Introduction

Hypergraph is a powerful tool in representing multiple relationships, which has many practical applications, such as cohesive mining [1], online groups [2] and neural networks [3]. In academic networks, the relations of publications and authors in DBLP can be effectively modeled as a hypergraph, where hyperedges correspond to academic papers and vertices represent authors [4]. Similarly, in protein complex networks, the proteins are mapped to vertices and protein complexes to hyperedges [5]. Contrary to traditional graphs which consist of pairwise connections among vertices, hypergraphs afford the capability of including two or more vertices within a single hyperedge. This richer structure enables the encoding of more complex relationships, thereby providing a more comprehensive representation of the data. Such advantages have fueled extensive developments in hypergraph processing tasks, designed to explore and analyze the unique properties of hypergraphs. For example, a novel hypergraph clustering method [6] is proposed that considers second or higher-order affinities between sets of data points, which is applicable to higher-order assignment problems and image segmentation. And parallel aggregated ordered hypergraphs [7] is to provide a highly readable representation of dynamic hypergraphs.

As the data scale is growing nowadays, the size of a hypergraph easily reaches billions of vertices and hyperedges, leading to prohibitively high computation costs. To address this, researchers have explored various acceleration methods like parallelism processing [8], pruning redundant hyperedges and incident vertices [9], and separating workloads in a distributed system [10]. However, the applications of these methods require consideration of some issues in conjunction with the algorithm itself, such as the parts available for parallelization, useless hyperedge vertices, or communication between clusters. This makes it difficult to apply these methods directly to arbitrary hypergraph processing. Hence, in this paper, we propose an underlying acceleration strategy that involves compression and hypergraph reordering, aimed at improving the efficiency of memory access in hypergraph processing to achieve acceleration, which can be directly applied to various hypergraph processing without considering the algorithm itself.

Hypergraphs are commonly represented by adjacency lists, where each entry in a vertex array is linked to its incident hyperedges and each entry in a hyperedge array is also linked to its vertices. Fig. 1(a) presents an example hypergraph of 8 vertices with 4 hyperedges and Fig. 1(b) is the corresponding adjacency list consisting of a vertex array and a hyperedge array. When traversing a hypergraph, we follow a pattern of interleaving the accesses to a vertex and a hyperedge. For instance, when we want to access the neighbor v_7 of v_6 in Fig. 1, the first step is to find e_1 in v_6's vertex array. Then, we need to access e_1's hyperedge array to find v_7. This is recognized as a memory-intensive workload. Meanwhile, such an adjacency list exposes poor spatial and temporal locality in the hypergraph traversal. When sequentially checking the neighbors of a vertex/hyperedge, the memory accesses are scattered to addresses distant from each other. Additionally, visiting the same neighbor of different vertices/hyperedges is directed to the redundant

- *Yu Liu, Qi Luo (corresponding author), Mengbai Xiao, Dongxiao Yu and Xiuzhen Cheng are with the School of Computer Science and Technology, Shandong University, Qingdao 266237, China (e-mail: {yuliu, luoqi2018}@mail.sdu.edu.cn; {xiaomb, dxyu, xzcheng}@sdu.edu.cn;).*
- *Huashan Chen is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China (e-mail: chenhuashan@iie.ac.cn).*



(a) A hypergraph G (b) The adjacency lists of G

Fig. 1. An example of a hypergraph and its adjacency lists.

parts in different linked lists. When accessing the incident hyperedges of v_0 and v_7 in Fig. 1, we need to access two different lists. However, in reality, they represent the same incident hyperedges. Such operations can hardly be cache-friendly and make the performance of traversal operations inferior.

Wei et al. [11] have implemented a one-dimensional array to store pairwise graphs and utilized graph reordering to rearrange the data, thereby improving data locality and accelerating graph traversal processing. Nevertheless, [11] only considers using one-dimensional arrays to store the graph without taking into account the removal of redundant parts in the graph. Additionally, compared to traditional pairwise graphs, hypergraphs can represent more information, making their reordering rules more complex and requiring a specific understanding of hypergraph characteristics. Considering the characteristics of multiple hyperedges interaction and unrestricted hyperedge sizes, we comprehensively evaluate the localities between vertices and hyperedges from both the vertex and hyperedge perspectives, which surpasses [11] that only consider the locality of vertices, thereby exhibiting a more significant superiority in enhancing data locality within hypergraphs. Finally, [11] did not consider the case of dynamic hypergraphs, whereas dynamic hypergraphs are more common in the real world. Motivated by the above, we plan to employ a similar method to enhance the efficiency of our traversal operations in hypergraphs. Storing neighbors of the vertices/hyperedges in a one-dimensional array is appealing, since traversing neighbors of vertices/hyperedges is materialized as accesses to consecutive memory addresses. Furthermore, as two consecutive entries might have common neighbors, duplicate ones could be merged to improve the temporal locality. Nonetheless, challenges exist in implementing such a data structure for hypergraphs.

Challenge 1. The order of entries matters. The one-dimensional array allows for arbitrary storage of vertices or hyperedges without any specific order. As consecutively storing two vertices without common neighboring hyperedges could hardly exploit the temporal locality, it is desired to find an ordering means so that for a hypergraph, redundant neighbors of consecutive entries are maximized. The similar idea, namely graph ordering, is also developed for pairwise graphs [11]. However, directly employing this method is not possible, since hypergraphs can contain hyperedges with more than two vertices. Simply applying graph reordering algorithms in [11] to hypergraphs can result in the loss of hypergraph-specific property, i.e., multiple hyperedges, leading to a significant reduction in the efficiency of hypergraph reordering.

Challenge 2. Updating a compression-array hypergraph is hard. A hypergraph could be evolving. For instance, authors are joining DBLP and new publications are available on a daily basis. Since the order of entities depends on the vertex-edge relationship in the hypergraph, adding or removing an entry might change the order suboptimal. If we run the hypergraph reordering algorithms from scratch to handle hypergraph updates, which proves to be a laborious endeavor for large-scale hypergraphs update.

In this paper, we propose a compression-array acceleration structure for hypergraphs so that the traversal operations become cache-friendly. To explore the order that maximizes the redundant neighbors of consecutive entries, we introduce a new metric called closeness, quantifying the locality between vertex pairs. This metric captures vertex/hyperedge locality so that we expect to gain the optimal order by maximizing closeness of consecutive entries. Vertex metrics should consider several factors such as neighboring relationships, two-hop reachability, and other local variables. Hyperedge metrics should consider various factors, including the vertices shared between hyperedges. Given that we prove the computation of the maximum closeness for the entire sequence to be an NP-hard problem, we have proposed an $\frac{1}{2w}$-approximation algorithm that leverages sliding windows and priority queues. For updates, we notice that only a subset of vertices/hyperedges are affected with respect to their closeness. By identifying the upper and the lower bounds of affected entries, the updates are constrained to a limited scope, alleviating the maintenance overhead of hypergraph. Then, a static hypergraph reordering algorithm is employed to conduct localized updates.

We summarize the contributions of this paper as follows:

- We develop an array-based acceleration structure to improve locality of hypergraph traversal, which is the essential operation to numerous hypergraph algorithms.
- We introduce a well-designed metric, namely closeness, and based on it, propose an NP-hard algorithm as well as a greedy one to reorder entries in our data structure to further exploit the temporal locality.

- We design a low-cost algorithm that updates our ordering result while maintaining the entry order still optimal.
- Experiments conducted on eight real-world datasets show that our design accelerate the hypergraph processing by up to 110%, reduces the CPU cache misses by up to 10% and saves up to 15% memory footprint. Moreover, the maintenance algorithm updates the hypergraph efficiently, taking only 30%-50% of the time of the reordering algorithm. We also integrate our method into Hygra [12], a well-known framework for processing hypergraphs. The performance of Hygra is improved by up to 60%, which further proves the effectiveness of our method.

Roadmap. Our paper is organized as follows: Section 2 reviews related work. Section 3 presents notions and our design of a compression-array acceleration structure for hypergraphs. Section 4 describes the algorithms that reorder the entries of a hypergraph. In Section 5, the maintenance algorithm is presented. We evaluate our algorithms in Section 6 and conclude the work in Section 7.

2 Related Work

In this section, we present a review of related work in locality optimization and graph processing acceleration, respectively

Locality optimization. Several studies have revealed that high hit rates in CPU cache effectively accelerate program execution [13]–[16]. Jo et al. [17] proposed the breadth-first (BF) data layout for single-machine based graph engines, which optimizes vertex access by storing processed vertices together in adjacent storage space. Photon [18] is a distributed graph processing framework that uses Property View for storing properties and an edge-centric execution engine with Hilbert-Order to improve locality. Chen et al. [19] proposed a heuristic algorithm for optimal data placement in data centers to minimize global data access cost. Hua et al. [20] proposed MERCURY, a multilevel caching scheme for efficient data placement in cloud computing, leveraging a novel multicore-enabled locality-sensitive hashing (MC-LSH) approach for accurate similarity capture. Ghoting et al. [21] proposed a cache-conscious prefix tree to address the issues of inadequate data locality and low instruction level parallelism (ILP). Datta et al. [22] presented the Algorithm Cache Miss Priority CPU Scheduler (CMPCS), which reduces power consumption by more than 20 percent while considerably improving performance of execution time. In the graph computing system GraphLite, Niu et al. [23] exploited CPU cache prefetching technology. Noll et al. [24] introduced a cache allocation scheme and incorporated a cache partitioning mechanism into the execution engine of a commercial DBMS, improving its overall performance by up to 38%. Yang et al. [25] introduced the FlashMob system, which enhances cache and memory bandwidth utilization by making memory accesses more sequential and regular. In summary, various approaches to locality optimization have been presented, including specialized data structures, scheduling algorithms, data placement strategies, prefetching techniques, and memory access patterns. These strategies have demonstrated considerable success in improving cache utilization and overall system performance.

Graph processing acceleration. Accelerating graph processing has been well studied for decades. Lakhotia et al. [26] introduced a novel Graph Processing Over Partitions (GPOP) framework that employs a partition-centric paradigm to compute PageRank in a cache-efficient, scalable, and work-efficient manner. Their framework aims to enhance cache and memory efficiency for graph processing. Park et al. [27] presented a 6x faster and cache-free implementation of the Floyd-Warshall algorithm, which solved all essential shortest-path graph problems. Mukkara et al. [28] introduced CacheGuided Scheduling (CGS), a specialized engine, that aims at minimizing cache misses by dynamically scheduling graph processing jobs. Wei et al. [29] proposed a graph ordering Gorder approach that minimizes the CPU cache miss ratio by keeping frequently accessed nodes together. Zhao et al. [30] improved graph processing by smoothly feeding graph partitions to memory and the last-level cache. [31] proposed an in-memory static cache called BFS-Aware Static Cache (Basc), which accelerates BFS operations. In conclusion, these studies highlight a range of techniques for accelerating graph processing, from dynamic and partition-centric paradigms to cache optimization methods and specialized engines.

In the realm of graph processing acceleration, the aforementioned works have made significant contributions. However, these techniques cannot be directly applied to accelerate hypergraph processing due to multiple hyperedges and unrestricted hyperedge size. Therefore, new strategies and frameworks tailored for hypergraph processing, considering their unique characteristics and challenges, are needed. Hence, in this paper, we propose a compression-array acceleration structure to accelerate hypergraph processing.

3 Hypergraph Compression

In this section, we initially present a technique for the one-dimensional array-based storage of hypergraphs. Subsequently, we propose a hypergraph compression method that enhances data locality by merging redundant parts of the hypergraph.

3.1 Preliminaries

Throughout this paper, we are always processing an undirected and unweighted hypergraph $G = (V, E)$, where V represents the set of vertices and E represents the set of hyperedges. We denote the number of vertices and hyperedges as $n = |V|$ and $m = |E|$, respectively. Each hyperedge e in E consists of a set of vertices in V, that is, $e = \{u_0, u_1, \ldots, u_i | u_i \in V\}$. We use $N_G(u)$ to denote the set of adjacency hyperedges of a vertex $u \in G$, defined as $N_G(u) = \{e_0, e_1, \ldots, e_i | u \in e_i, e_i \in G\}$. Furthermore, the degree $d_G(u)$ of a vertex $u \in G$ is defined as $d_G(u) = |N_G(u)|$. Throughout this paper, when the context is clear, we ignore the input hypergraph's symbols, e.g., $N(u)$ instead of $N_G(u)$. The symbols used in this paper are summarized in Table 1.

TABLE 1
Symbols and Descriptions

Notation	Definition
$G = (V, E)$	an undirected and unweighted hypergraph
$N_G(u)$	the incident hyperedges of a vertex $u \in G$
$d_G(u)$	the degree of $u \in G$
$S(u, v)$	the closeness of the pair of vertices u, v
$\mathcal{L}$	the reordering sequence of vertices
$\delta(u)$	the index of the vertex u in the sequence
$F(\mathcal{L})$	the sum of the closeness of the vertex pairs in $\mathcal{L}$
$FV(u)$	the sum of the closeness of the vertex u in the sequence
$\phi(e)$	the set of neighboring vertices of the hyperedge e

3.2 Compression-Based Hypergraph Structure

The prevalent storage format for hypergraphs, the adjacency list, has been observed to exhibit suboptimal locality. In this subsection, we present to store a hypergraph with one-dimensional arrays. This provides an opportunity of merging the same neighbors of consecutive vertices or hyperedges stored in the array, thus improving the locality. Furthermore, we can explore how to order the entities in the array for maximizing the number of neighbors to be merged, which is discussed in Section 4.

After observing the hypergraph adjacency list presented in Fig. 1 (b), we have identified the neighboring lists are redundant. For instance, v_1 and v_2 shares the common neighbor of a hyperedge e_0, while two hyperedges, e_2 and e_3, both have common neighbors of v_0 and v_7. These redundant neighbors are stored and accessed at individual locations in the main memory, exhibiting poor locality in hypergraph traversals. Herein, we propose to store a hypergraph with one-dimensional arrays that strive to accelerate the hypergraph traversals.

Definition 1. (**Hypergraph Storage Structure**) Given a hypergraph $G = (V, E)$, we define two one-dimensional arrays to store the incident hyperedges and the vertices of hyperedges. For each vertex and hyperedge, we define two indexes (start index and end index) to determine its range in the one-dimensional arrays.

Since the ranges covering neighbors of a vertex/hyperedge could overlap with each other, the redundant neighbors of two consecutive entities will only show once.

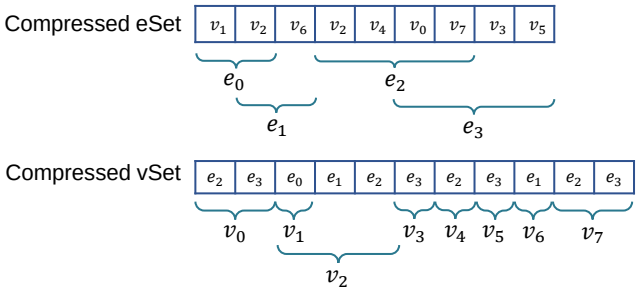


Fig. 2. The compression hypergraph structure of Fig. 1.

Example 1. Fig. 2 is our newly defined data structure, which represents the same hypergraph as Fig. 1. In $eSet$, the *hyperedge set*, a hyperedge is defined by a range of vertices continuously stored that are its neighbors. $vSet$, the *vertex set*, defines vertices in the same way. The ranges of consecutive hyperedges or vertices, like e_2 and e_3, are overlapped so that the redundant neighbors (v_0 and v_7) are stored only once.

To generate such a compressed data structure for a hypergraph, we employ a greedy approach shown in Algorithm 1. In addition to $eSet$ and $vSet$ that contain the complete neighbor information, we also have to define eID and vID, which are two arrays saving the ranges of neighbors for hyperedges and vertices, respectively. Specifically, $eID[i].s$ and $eID[i].d$ denote the start and end positions of the hyperedge e_i in $eSet$. Algorithm 1 compresses the hyperedges into $eSet$ and eID (Lines 1-17), and vertices into $vSet$ and vID (Line 18). For a hyperedge e_i, we denote e^*, e^- as $e_{i-1} \cap e_i, e_i \cap e_{i+1}$, respectively (Line 1-3). Then, it iterates over all hyperedges, distinguishing the last hyperedge from other since there is no hyperedge after the last hyperedge (Line 5-8). It finds overlapping parts between adjacent hyperedges and sorts vertices accordingly. We arrange the hyperedges e_i that do not intersect with e_{i-1} or e_{i+1} due to that they only appear once (Line 11-13). Then, we arrange the intersecting parts of e_i and e_{i+1}, noting that we should not arrange the intersecting parts of e_i and e_{i-1} included here. This is because they have already been arranged in the previous iteration (Line 14-16). In the same way, for a vertex v_i, its incident hyperedges are compressed similarly (Line 18).

Algorithm 1 Generate compressed hypergraphs

Input: $G = (V, E)$
Output: $eSet, eID, vSet, vID$
1: $eSet \leftarrow \emptyset$, $e^* \leftarrow \emptyset$, $e^- \leftarrow \emptyset$;
2: $eID[i].s \leftarrow 1$, $eID[i].e \leftarrow 1, \forall i \in 1, 2, \ldots, m$;
3: $cnt \leftarrow 1$;
4: **for** $i \leftarrow 1$ to m **do** ▷ *Compressing hyperedges.*
5: **if** $i = m$ **then**
6: **for** $u \in e_i \setminus e^-$ **do**
7: $eSet[cnt] \leftarrow u$, $cnt \leftarrow cnt + 1$;
8: $eID[i].e \leftarrow cnt$;
9: **else**
10: $e^* = e_i \cap e_{i+1}$;
11: **for** $u \in e_i \setminus \{e^- \cup e^*\}$ **do**
12: $eSet[cnt] \leftarrow u$, $cnt \leftarrow cnt + 1$;
13: $eID[i + 1].s \leftarrow cnt$;
14: **for** $u \in e^* \setminus e^-$ **do**
15: $eSet[cnt] \leftarrow u$, $cnt \leftarrow cnt + 1$;
16: $eID[i].e \leftarrow cnt$;
17: $e^- \leftarrow e^*$;
18: Compressing the incident hyperedges of vertices by a similar approach in Line 1-17;
19: **return** $eSet, eID, vSet, vID$;

Performance Analysis. To perform a time complexity analysis of Algorithm 1, we introduce the following symbols. Given a hypergraph $G = (V, E)$, we define $\bar{c}$ as the maximum cardinality of hyperedges and $\bar{d}$ as the maximum degree of vertices.

Theorem 1. The time complexity of Algorithm 1 is $O(n * \overline{d} + m * \overline{c})$ and its space complexity is $O(\overline{c})$.

Considering the limitation of space, detailed proofs of all theorems are given in Appendix of this paper.

4 HYPERGRAPH REORDERING

This section explores techniques for removing redundant components from hypergraphs. Fig. 2 shows that the incident hyperedges of vertices v_0 and v_7 are identical. Nevertheless, the redundancy is hard to eliminate because of their distinct affiliations with different parts in memory. Thus, we introduce the concept of hypergraph reordering, which entails rearranging the memory location of the hypergraph to remove the duplicated parts.

Vertices and hyperedges, although both fundamental components of hypergraphs, exhibit distinct characteristics. Specifically, vertices typically represent entities, while hyperedges represent complex many-to-many relationships among these entities. Therefore, their measures of closeness are inherently different. To ensure that vertices and hyperedges are reordered accurately, we choose different metrics to handle the vertex reordering and hyperedge reordering separately. Since the techniques utilized in both methods are identical, except for the initial function definitions, we will elaborate on the vertex reordering method as the main example.

4.1 Vertex Reordering Principles

The vertex reordering problem can be addressed via three steps: 1): Define the closeness between vertex pairs in the hypergraph. 2): Define the closeness of vertex sequence. 3): Develop an approximation algorithm that utilizes the results from steps 1 and 2 to find an optimal vertex sequence. Each step presents its own challenges, and their implementation will be discussed in detail.

In this subsection, our objective is to rearrange vertices in memory to enhance the storage locality within hypergraphs. The key to addressing this issue lies in devising a metric for quantifying the closeness of arbitrary vertex pairs. However, defining a suitable metric is immensely challenging due to the complex nature of hypergraphs. Compared to pairwise graphs, hypergraphs contain more information, necessitating the precise characterization of vertex pair closeness. To overcome this challenge, we introduce a metric proposed in Definition 2 based on the concept of neighbors and two-hop reachability, which are prevalent in hypergraph processing.

Subsequently, it becomes imperative to ascertain the maximal closeness of a vertex sequence. Thus, to improve the storage locality in hypergraphs, frequently accessed vertices ought to be situated adjacently in memory, enabling the CPU to retrieve the data in a single operation. Assumed that the CPU can retrieve w data in a single operation, only the closeness between a vertex and the w vertices preceding and succeeding it in the vertex sequence needs to be considered. This can be realized by defining the closeness of a vertex sequence, as explicated in Definition 3. It is vital to underscore that the vertex reordering algorithm bears no impact on the processing of hypergraphs, as it solely alters the arrangement of vertices in memory. Hence, this algorithm can expedite the processing of various hypergraphs.

Below, we will provide an exhaustive description of each stage in the procedure. Initially, we introduce the concept of vertex pair closeness, which quantifies the locality between any two vertices within the hypergraph. The vertex pair closeness measure is instrumental in our vertex reordering approach, as it facilitates the grouping of frequently accessed vertices to enhance data locality. The formal definition of vertex pair closeness is presented below.

Definition 2. (**Vertex Pair Closeness**) Given a hypergraph $G = (V, E)$ and two vertices $u, v \in V$, we define the closeness between u and v as $S(u, v)$, which is formally defined as follows.

$$S(u, v) = S_c(u, v) + S_p(u, v) \tag{1}$$

We denote $S_c(u, v)$ as the number of common hyperedges between vertices u and v, where $S_c(u, v) = |N_G(u) \cap N_G(v)|$. We define $p(u, v)$ as a path of length two between u and v, where $p(u, v) = u \to e_1 \to x \to e_2 \to v$ with $u \neq v \neq x, e_1 \neq e_2$, and u, v, x, e_1, e_2 are all vertices in the hypergraph G. Then, we define $S_p(u, v)$ as the number of paths between u and v that satisfy the definition of $p(u, v)$.

In the foregoing discussion, we consider two scenarios: (1) a pair of vertices coexist within hyperedges; (2) a pair of vertices are reachable through a path of length two.

1) The coexistence of vertex pairs in multi-hyperedges serves as a correlation measure, revealing the interconnected characteristics of these vertex pairs. This dimensionality diversity has significant practical value in multiple real-world application scenarios, such as social network analysis and medical networks.
2) By considering paths of length two, we introduce neighborhood consistency. This design is based on the observation that vertex pairs connected through paths of length two generally share similar neighborhood structures. Such neighborhood consistency not only effectively measures structural similarity but also identifies vertex pairs that act as bridges between different social groups or network modules.

By integrating both factors, we provide a more comprehensive measure of closeness between vertex pairs, as opposed to relying on a single factor. We further consider the vertex sequence closeness based on the closeness of vertex pairs. While, the data a CPU can retrieve in a single operator is constrained. We introduce a parameter w, indicating the size of data that can be fetched by the CPU. By accounting these factors, it becomes necessary to compute the closeness between a vertex and the w vertices that precede and succeed it in the sequence. We formalize this concept through a sequence closeness equation, which aggregates the closeness of all vertex pairs within the sequence.

Definition 3. (**Sequence Closeness**) Given a hypergraph $G = (V, E)$ and a sequence $\mathcal{L}$ of all vertices in G, $F(\mathcal{L})$ is denoted as closeness of vertex sequence.

$$F(\mathcal{L}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{\min\{i+w-1, n\}} S(u_i, u_j), w \geq 2 \tag{2}$$

u_i, u_j denote the i-th, j-th vertices in the sequence $\mathcal{L}$ and $w(w \geq 2)$ is the size of the data that can be fetched by the CPU.

In Equation 2, it becomes evident that the closeness of vertex pairs can be determined exclusively for a sliding window of length w. Hence, we can efficiently compute the closeness of all vertex pairs in a sequence by advancing the sliding window of size w from the beginning to the end, while accumulating the closeness of pairs encompassed within the window.

For convenience, we define $\delta(u)$ for $u \in V$ as the index of vertex u in sequence $\mathcal{L}$. Below, we provide the formulation of our problem.

Problem 1. (**Optimal Sequence**) Given a hypergraph $G = (V, E)$, find an optimal sequence $\mathcal{L}$ that maximizes $F(\mathcal{L})$.

4.2 Hyperedge Reordering Principles

In this subsection, we present a definition for hyperedge closeness. It is worth noting that the technique used for reordering hyperedges is congruent to that used for reordering vertices. Further elaboration on this problem is deemed unnecessary.

Definition 4. (**Hyperedge Pair Closeness**) Given a hypergraph $G = (V, E)$ and two hyperedges $e_0, e_1 \in E$, we define the closeness between e_0 and e_1 as $S(e_0, e_1)$, which is formally defined as follows.

$$S(e_0, e_1) = |\{u | u \in e_0 \& u \in e_1\}| \tag{3}$$

The closeness between two hyperedges is defined as the number of shared vertices.

4.3 Greed-based algorithms

In this subsection, we discuss the development of efficient algorithms to tackle the hypergraph reordering problem for large-scale hypergraphs. Enumerating all possible sequences to find an optimal one is time-consuming. This is due to the fact that the problem is NP-hard, as demonstrated in our proof of Problem 1. To address this, we propose an approximation algorithm that focuses on the maximum closeness of the vertex sequence, where we only need to compute the closeness between it and the surrounding vertices for each vertex. Our algorithm incorporates vertices with the highest closeness into the sequence, yielding a more effective and efficient solution.

First, we prove that Problem 1 is NP-hard. This emphasizes the difficulty in selecting an optimal vertex sequence and accentuates the necessity for more efficient algorithms to tackle this issue.

Theorem 2. Given a hypergraph $G = (V, E)$, finding an optimal sequence $\mathcal{L}$ to maximize $F(\mathcal{L})$ is NP-hard.

Considering that Problem 1 is NP-hard, it is infeasible to compute an exact solution within polynomial time. To surmount this constraint, we introduce an approximation algorithm that offers an efficient solution for identifying the optimal vertex sequence. Our strategy greedily inserts a vertex v to maximize the sum of $S(u, v)$ between v and the preceding $w-1$ vertices u of v. We maintain a sequence $\mathcal{L}$ of length n, where $\mathcal{L}[i]$ signifies the vertex represented by the i-th position in the sequence $\mathcal{L}$. We employ V_c to designate the set of candidate vertices that have not yet been allocated to $\mathcal{L}$. To ascertain the position of a vertex v in the sequence $\mathcal{L}$, we compute the sum of $S(u, v)$ for $u \in \mathcal{L}[i-w+1, i-1]$ for all $v \in V_c$. The vertex v that maximizes this sum is incorporated into the i-th position of the sequence. We formalize this approach in the subsequent expression:

$$FV(v) = \sum_{j=max\{1,i-w+1\}}^{i-1} S(u_j, v), u_j \in \mathcal{L}[i-w+1, i-1] \tag{4}$$

Our algorithm focuses on identifying a vertex $v \in V_c$ that maximizes $FV(v)$, representing the sum of $S(u, v)$ between v and the previous $w-1$ vertices u of v. Subsequently, we develop Algorithm 2 based on this methodology.

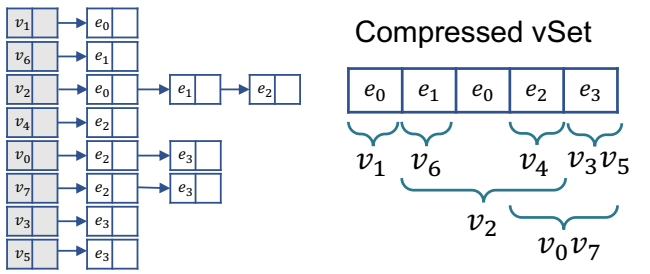
For the sake of efficient computation, we employ a priority queue approach that preserves the $FV(\cdot)$ values of the vertices in V_c (Line 1). We initiate the process by selecting the vertex with the highest degree in the hypergraph as the starting vertex (Line 3-5). This vertex is presented in the most common hyperedges with other vertices and can reach more vertices through paths of length two. As vertices are allocated to the sequence $\mathcal{L}$, we increase the $FV(\cdot)$ of their associated vertices (Line 8-14) according to Equation 4. Given that the size of the sliding window is w, the insertion of a vertex results in the leftmost vertex exiting the sliding window. To account for this change, we decrease the $FV(\cdot)$ of the vertices associated with $\mathcal{L}[i-w]$ (Line 16-22). Then, we select the vertex with the maximum $FV(\cdot)$ value from V_c and incorporate it into $\mathcal{L}$ (Line 23-26). This operation is reiterated until V_c is empty (Line 7).

Performance Analysis. To perform a time complexity analysis of Algorithm 2, we introduce the following symbols.

Let $G = (V, E)$ be a hypergraph, where V is the set of vertices and E is the set of hyperedges. We represent the number of multiple neighbors of a vertex $u \in V$ as $mdeg(u)$, defined as the sum of the sizes of the hyperedges containing u, i.e., $mdeg(u) = \sum_{e \in N_G(u)} |e|$.

Theorem 3. For a hypergraph $G = (V, E)$, the time complexity of Algorithm 2 is $O(\sum_{u \in V} (mdeg(u))^2 + n^2)$.

Theorem 4. Algorithm 2 give $\frac{1}{2w}$-approximation for finding an optimal sequence $\mathcal{L}$ that maximizes $F(\mathcal{L})$.



(a) The reordered adjacency list (b) The reordered compression eSet

Fig. 3. The result of reordering the vertices of the hypergraph in Fig. 1.

Algorithm 2 Find an optimal sequence

Input: $G = (V, E)$
Output: $\mathcal{L}$

$FV[u] \leftarrow 0$ for all $u \in V$;
$V_c \leftarrow V$;
$u^* \leftarrow \arg\max_{u \in V_c} d(u)$;
$V_c \leftarrow V_c \setminus u*$;
$\mathcal{L}[1] \leftarrow u*$;
$i \leftarrow 2$;
while V_c is not empty **do**
  $v_{insert} \leftarrow \mathcal{L}[i-1]$;
  for $e_1 \in N(v_{insert})$ **do**
    for $u \in e_1$ & $u \in V_c$ **do**
      $FV[u] \leftarrow FV[u] + 1$;
      for $e_2 \in N(u)$ & $e_2 \neq e_1$ **do**
        for $v \in e_2$ & $v \in V_c$ & $v \neq u$ **do**
          $FV[v] \leftarrow FV[v] + 1$;
  if $i > w$ **then**
    $v_{delete} \leftarrow \mathcal{L}[i-w]$;
    for $e_1 \in N(v_{delete})$ **do**
      for $u \in e_1$ & $u \in V_c$ **do**
        $FV[u] \leftarrow FV[u] - 1$;
        for $e_2 \in N(u)$ & $e_2 \neq e_1$ **do**
          for $v \in e_2$ & $v \in V_c$ & $v \neq u$ **do**
            $FV[v] \leftarrow FV[v] - 1$;
  $u^+ \leftarrow \arg\max_{u \in V_c} FV[u]$;
  $\mathcal{L}[i] \leftarrow u^+$;
  $V_c \leftarrow V_c \setminus u^+$;
  $i \leftarrow i + 1$;
return $\mathcal{L}$;

Example 2. Fig. 3 shows the results of vertex reordering in the hypergraph illustrated in Fig. 1. The final $eSet$ representation indicates that multiple overlapping parts have been eliminated. In Fig. 3(a), Algorithm 2 is initially employed to compute the optimal vertex sequence for all vertices, which is $\{v_1, v_6, v_2, v_4, v_0, v_7, v_3, v_5\}$. This optimal sequence is sorted based on the Vertex Pair Closeness metric among the vertices. Subsequently, Algorithm 1 is utilized to compress the hypergraph according to this optimal sequence. This process has removed most repetitive parts in the hypergraph, leading to a substantially improved locality.

Discussions of Applications. Our hypergraph reordering technique has broad applicability, enhancing not only the execution efficiency of memory algorithms but also enhancing the performance of external-storage algorithms. Specifically, by meticulously reordering the hypergraph structure in external storage, we ensure that vertices and hyperedges accessed consecutively are physically adjacent in the external storage layer. This external storage optimization significantly reduces the frequency of data retrieval from external storage, and I/O costs during the algorithm's execution, thereby elevating the overall efficiency of the algorithm.

5 Maintenance the optimal sequence for dynamic hypergraphs

In the real world, hypergraphs are constantly evolving. If the reordering algorithm must be executed from scratch every time the hypergraph is updated, it would be unbearable. In this section, our elaboration is exemplified by maintaining vertices as a primary case. Since the process of maintaining hyperedges is similar to that of maintaining vertices, we avoid duplicating the discussion on hyperedge maintenance. Hence, it is important to maintain the vertex sequence. However, discerning the vertices in a sequence that have changed is challenging. After the insertion or deletion of a hyperedge from a hypergraph, the precise identification of the modified vertices within the sequence remains unattainable. We acknowledge that the insertion or deletion of a hyperedge from a hypergraph results in a specific range of vertices within the sequence changing. Consequently, we merely require the recognition of the lower and upper boundaries of the variation range, followed by the execution of the static algorithm for local amendments. Predicated upon this observation, we propose a local update algorithm for maintaining the sequence $\mathcal{L}$ in dynamic hypergraphs.

Initially, let us define a hyperedge-neighbor, denominated as $\phi(e)$, representing the set of incident vertices pertaining to a given hyperedge e.

Definition 5. (**hyperedge-neighbor**) We denote the hyperedge-neighbor of e as $\phi(e)$, which is defined by the set of vertices $v' \in e'$ that belong to the hyperedges $e' \in N_G(v)$ containing a vertex $v \in e$, i.e.,

$$\phi(e) = \{v' | v' \in e', e' \in N_G(v), v \in e\} \tag{5}$$

Building upon the previously defined hyperedge neighbors $\phi(e)$, we propose a pivotal theorem for vertices whose $FV(\cdot)$ undergoes a modification.

Theorem 5. Given a hypergraph $G = (V, E)$ and a hyperedge e, when the hyperedge e is inserted into or deleted from the hypergraph G, $FV(u)$ changes only if $u \in \phi(e)$, for any vertex u.

As we previously demonstrated, $FV(u)$ of a vertex $u \in \phi(e)$ may undergo modifications upon the insertion or deletion of a hyperedge e in a hypergraph. In consideration of this, we present Theorem 6, which identifies the portions of the sequence $\mathcal{L}$ that may undergo changes following the insertion or deletion of a hyperedge. Theorem 6 is employed in Algorithm 3.

Theorem 6. When a hyperedge e is either inserted into or deleted from the hypergraph $G = (V, E)$, it follows that any vertex located within the sequence $\mathcal{L}[lb, ub]$ is susceptible to modifications. The sequence $\mathcal{L}$ represents the original sequence, while $\mathcal{L}'$ denotes the updated sequence. Subsequently, we define lb and ub as follows:

1) $lb = \min\{\delta(u) | u \in \phi(e)\} + 1$.
2) $\mathcal{L}[lb, ub]$ contains all the vertices in $\phi(e)$ and $\mathcal{L}[ub - w + 1, ub] = \mathcal{L}'[ub - w + 1, ub]$.

Drawing upon the aforementioned theorems, we propose a rigorous maintenance algorithm for the sequence $\mathcal{L}$ in dynamic hypergraphs, which is exhibited in Algorithm 3.

Algorithm 3 Optimal Sequence Maintenance

Input: $G = (V, E)$, $\mathcal{L}$, e
Output: $\mathcal{L}$;
1: $\mathcal{L}' \leftarrow \mathcal{L}$;
2: $V_t \leftarrow \phi(e)$;
3: $vs \leftarrow \arg\min_{u \in V_t} \delta(u)$;
4: $V_t \leftarrow V_t \setminus vs$;
5: $lb \leftarrow \delta(vs)$;
6: $V_c \leftarrow \{u | \delta(u) > \delta(vs)\}$;
7: $FV[u] \leftarrow 0, \forall u \in V_c$;
8: **for** $u \in \mathcal{L}[lb - w + 2, lb]$ **do**
9: $\quad vi \leftarrow u$;
10: $\quad$ Line 9-14 of Algorithm 2;
11: $p \leftarrow lb + 1$;
12: **while** V_c is not empty **do**
13: $\quad u^+ \leftarrow \arg\max_{u \in V_c} F[u]$;
14: $\quad \mathcal{L}'[p] = u^+, p \leftarrow p + 1$;
15: $\quad V_c \leftarrow V_c \setminus u^+, V_t \leftarrow V_t \setminus u^+$;
16: $\quad vi \leftarrow \mathcal{L}'[p - 1]$;
17: $\quad$ Line 9-14 of Algorithm 2;
18: $\quad$ **if** $i > w$ **then**
19: $\quad\quad vd \leftarrow \mathcal{L}'[p - w]$;
20: $\quad\quad$ Line 17-22 of Algorithm 2;
21: $\quad$ **if** $V_t = \emptyset$ and $\mathcal{L}'[p - w, p - 1] = \mathcal{L}[p - w, p - 1]$ **then**
22: $\quad\quad$ **break**;
23: $\mathcal{L} \leftarrow \mathcal{L}'$;
24: **return** $\mathcal{L}$;

To facilitate the subsequent algorithmic discussion, we define a set of notations. Let $\mathcal{L}$ denote the original optimal sequence, and $\mathcal{L}'$ represent the optimal sequence obtained after the insertion or deletion of a hyperedge. V_t is defined as the set of vertices whose $FV(\cdot)$ changes when a hyperedge e is inserted or removed. V_c is used to represent the set of vertices that follow vertex vs in the sequence $\mathcal{L}$. When V_c is empty, it indicates that the sequence $\mathcal{L}$ has been fully traversed, thus terminating the algorithm. In accordance with the proof of Theorem 6, the update of sequence $\mathcal{L}$ initiates from position lb. To ascertain the value of the starting position lb, we compute it first (Lines 2-5). We embrace the same update strategy as Algorithm 2, specifically, selecting the vertex exhibiting the maximal $FV(\cdot)$ value for integration into the extant sequence $\mathcal{L}$. To identify the vertex with the maximum $FV(\cdot)$ value, we compute the $FV(\cdot)$ values of vertices within $\mathcal{L}[lb - w + 2, lb]$ (Line 8-10). Subsequently, we incorporate the vertex with the maximal $FV(\cdot)$ value into the new sequence $\mathcal{L}'$, updating the $FV(\cdot)$ value of corresponding vertices (Lines 13-20). We reiterate this step until all vertices in every V_t have been amalgamated into the new sequence $\mathcal{L}'$. Next, we contrast the new sequence $\mathcal{L}'$ with the original sequence $\mathcal{L}$ to discern whether they possess an identical sequence of length w, namely, $\mathcal{L}[ub - w + 1, ub] = \mathcal{L}'[ub - w + 1, ub]$ (Lines 21-22). If such an identical sequence of length w prevails, all vertices have undergone updates upon the insertion of hyperedge e, culminating in the algorithm's conclusion, consistent with Theorem 6. Conversely, if no identical sequence of length w is detected, we must persist in selecting the vertex with the maximal $FV(\cdot)$ value until the $FV(\cdot)$ of all vertices in G have been updated.

Performance Analysis. To perform a time complexity analysis of Algorithm 3, we introduce the following symbols.

Let $G = (V, E)$ be a hypergraph. For a vertex $u \in G$, the number of multiple neighbors is denoted by $mdeg(u)$ and is defined as the sum of the size of all hyperedges that contain u, i.e., $mdeg(u) = \sum_{e \in N_G(u)} |e|$. We assume that $maxD$ is defined as the maximum value of $\sum_{v \in e, e \in N_G(u)} mdeg(v)$ for all vertices $v \in G$, i.e., $maxD = \max\{\sum_{v \in e, e \in N_G(u)} mdeg(v), \forall v \in G\}$.

Theorem 7. Given a hypergraph $G = (V, E)$ and a hyperedge e, the worst-case time complexity of Algorithm 3 is $O(\sum_{u \in V} (mdeg(u))^2 + n^2)$.

6 EXPERIMENT

In this section, we evaluate our algorithms and present experimental results. We start with a description of the datasets and the environment settings.

Datasets. We evaluate our algorithms on eight distinct datasets, which are downloaded from ARB[1]. The TaAU dataset [32] is a temporal higher-order network where vertices are tags assigned to askubuntu.com questions, and hyperedges represent sets of tags for each question. The MaAn dataset [33] consists of a hypergraph with vertices as tags in Math Overflow questions and hyperedges as sets of answered questions. The WaTr dataset [34] is a hypergraph with vertices as products and hyperedges as sets of co-purchased products at Walmart. The ThAU and ThMS datasets [32] are temporal higher-order networks with vertices as users on askubuntu.com and math.stackexchange.com, respectively, and hyperedges as user participation in threads within a 24-hour duration. The CoMH and CoGe datasets [32], [35] are temporal higher-order networks, where vertices are authors and hyperedges are publications in the "History" and "Geology" fields of the Microsoft Academic Graph. The CoDB dataset [32] is a temporal higher-order network featuring vertices as authors and hyperedges as publications documented on DBLP.

We remove all island vertices in the datasets. The basic statistics of all datasets are presented in Table 2, where d_{max} is the maximum degree of all vertices, and c_{max} is the maximum cardinality of all hyperedges.

TABLE 2
Real-World Hypergraph Datasets

Dataset	$\|V\|$	$\|E\|$	d_{max}	c_{max}
TaAU	3,021	219,076	19,631	5
MaAn	73,851	5,446	173	1,784
WaTr	88,860	69,906	5,733	25
ThAU	90,054	117,764	2,247	14
ThMS	153,806	563,710	12,403	21
CoMH	503,868	308,934	1,077	925
CoGe	1,091,979	1,045,462	1,125	284
CoDB	1,836,596	2,955,129	1,399	280

1. https://www.cs.cornell.edu/~arb/data/

Settings. All of our algorithms are implemented in C++ and compiled using GUN GCC 11.3.0. We conduct all experiments on a machine with an AMD Ryzen Threadripper PRO 5995WX 64-Cores 2.7 GHz CPU and 256 GB memory, running Ubuntu 22.04.1 LTS. The machine has an L1 cache size of 2 MiB, an L2 cache size of 32 MiB, and an L3 cache size of 356 MiB.

Parameters. We define the width of the sliding window w as 16. This value matches the number of data elements (4 bytes) one cache line (64 bytes) could hold on our machine, which we determined based on the cache line size of our machine (64 bytes) and the size of an integer data type (4 bytes). Thus, we can deduce that the machine is capable of accessing a total of 16 individual data elements within a single operator.

Algorithms. We enumerate four hypergraph acceleration structures optimized by our algorithms and the adjacency list as the baseline, which is shown in Table 3.

TABLE 3
Hypergraph Storage Structures

Name	Storage Format	Reordering
Array-V&E	one-dimensional array	reordering of vertex and hyperedge
Array-V	one-dimensional array	reordering of vertex
Array-E	one-dimensional array	reordering of hyperedge
Array-vanilla	one-dimensional array	no reordering
Adjlist	adjacent list	no reordering

Hypergraph processing tasks. To comprehensively evaluate the proposed algorithms, we carry out experiments with eight widely adopted hypergraph processing tasks. These hypergraph processing methods are presented below.

- Neighbors Query (NQ) [11]
- Breath-First Search (BFS) [36]
- Depth-First Search (DFS) [36]
- Strongly Connected Component (SCC) [12]
- PageRank (PR) [12]
- Hypergraph Decomposition (KCore) [2]
- Single-Source Shortest Paths (SSSP) [37]
- Maximal Independent Set (MIS) [38]

For NQ, BFS, and DFS, we randomly select 1000 vertices. For SCC, KCore, and MIS, we repeat the experiments 1000 times. For SSSP, we randomly select 1000 pairs of vertices to compute the shortest path. We report the mean of the experimental results.

Experimental goals. To assess the performance and viability of our algorithms, we design seven experiments to demonstrate the superiority of our algorithms:

- **Experiment** 6.1: Demonstrate the effectiveness of hypergraph compression and reordering in performance optimization by speedup ratio.
- **Experiment** 6.2: Demonstrate that cache miss ratio is a key factor affecting the algorithm's performance.
- **Experiment** 6.3: Substantiate the rationale and significance of vertex pair closeness we define.
- **Experiment** 6.4: Evaluate the computational efficiency of the maintenance algorithm.
- **Experiment** 6.5: Quantify the contribution of hypergraph compression and reordering to compression efficiency.
- **Experiment** 6.6: Demonstrate that hypergraph reordering strategies can be easily integrated into existing hypergraph processing frameworks and achieve excellent results.
- **Experiment** 6.7: Provide a detailed evaluation of multiple optimization metrics for hypergraph compression and reordering algorithms on specific datasets.

Reproducibility. The source code of this paper is released at https://github.com/muranhuli/Reordering-and-Compression-for-Hypergraph-Analysis.

6.1 Algorithm Speedup Ratio

We first evaluate the speedup ratios of the acceleration structures over the adjacency list, and the results are presented in Fig. 4. We can clearly observe that data storage structures Array-V&E, Array-V, Array-E, and Array-vanilla consistently outperform the data storage structures Adjlist in all experiments. In particular, data storage structure Array-vanilla speeds up the hypergraph processing methods by 10%-30%. This indicates that when processing hypergraphs, switching from the adjacency list to the one-dimensional array substantially improves the performance. On the other hand, the speedup ratios of Array-V and Array-E show that adding our reordering algorithm further improves the hypergraph processing by 10%-90%. By rearranging the order of vertices and hyperedges, the hypergraph processing methods have higher opportunities for reading data from the cache, thus achieving higher speedups. Simultaneously reordering vertices and hyperedges reaches the speedup ratios of 20%-110%. More interestingly, we observe that even in the scenarios where Array-V, Array-E, and Array-vanilla optimizations were not so effective, such as with the BFS algorithm of ThMS and the SSSP algorithm of CoDB, data storage structure Array-V&E still yields high. These results justify the design of our acceleration structure, which could speed up a broad range of hypergraph processing methods substantially.

6.2 Cache Miss Ratio

We also measure the cache miss ratio of different hypergraph processing methods running over various data structures. A low cache miss ratio reflects our design is cache-friendly. In the experiments, we measure L1 cache miss ratios by the Linux tool perf[2]. For each hypergraph processing method, we aggregate the cache miss ratios of different datasets and report the results in Fig. 5. The box plot consists of five components: the lower whisker (first quartile), lower hinge (25th percentile), median (50th percentile), upper hinge (75th percentile), and upper whisker (maximum value).

Fig. 5 displays that the overall cache miss ratio of Array-V&E, Array-V, Array-E, Array-vanilla, and Adjlist manifests a decreasing trend, indicating that our proposed algorithm improves the data locality in hypergraphs effectively. This also aligns the speedup ratios measured in Section 6.1, which confirms that the speedup largely results from the

2. https://perf.wiki.kernel.org/index.php/Main_Page

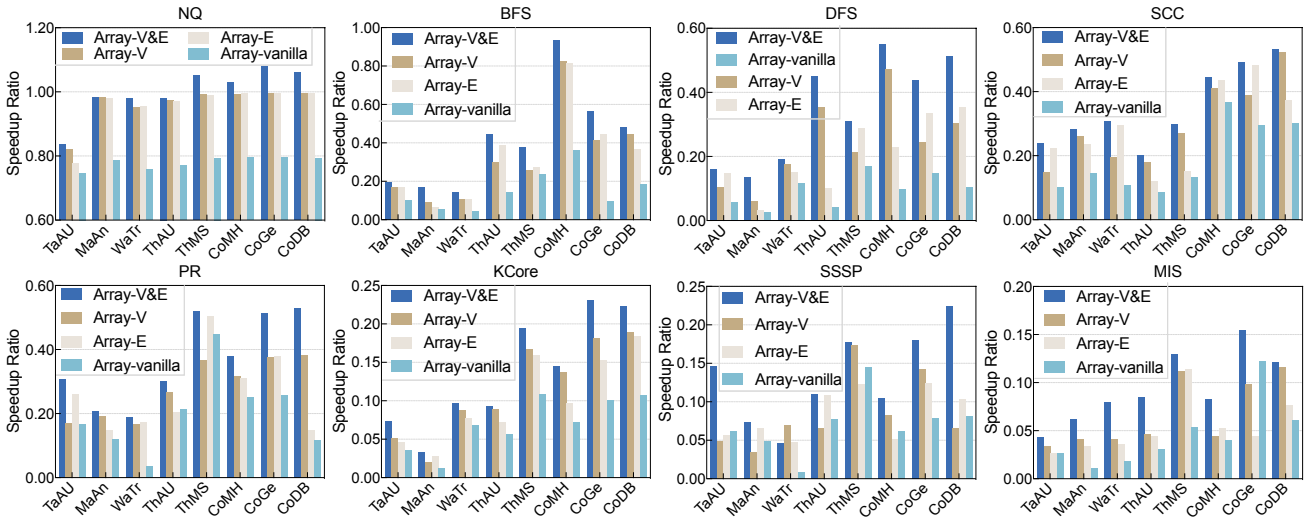


Fig. 4. Speedup ratios of four optimized data structures.

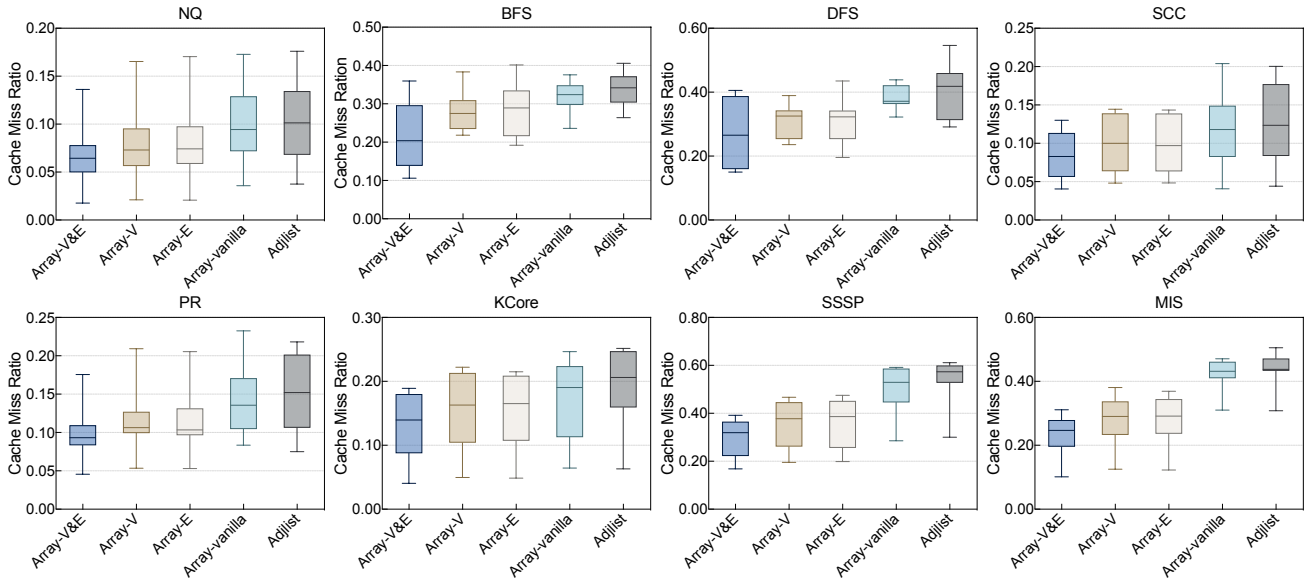


Fig. 5. L1 miss ratio of five data structures.

improved data locality. However, we note that in certain experiments, such as the MIS experiment of Array-vanilla, the cache miss ratio does not decrease significantly relative to the MIS experiment of Adjlist. Nevertheless, the execution time of the MIS experiment of Array-vanilla is reduced relative to the MIS experiment of Adjlist, as illustrated in Figure 4. This phenomenon can be ascribed to the fact that the miss rate is computed by dividing the number of misses by the aggregate quantity of references. Consequently, when both the number of misses and the total count of references decline, the miss rate may rise rather than decrease. The program's running time can be expressed as follows:

$$t = t_c * n_c + t_m * n_m \tag{6}$$

Here, t denotes the total time, t_c represents the time required to access when the cache is hit, t_m is the time required to access when the cache is missed, n_c is the number of cache hits, and n_m is the number of cache misses. When both n_c and n_m decrease, the total running time of the program will decrease, even if the cache miss rate $\frac{n_m}{n_c+n_m}$ does not necessarily decrease.

6.3 Ablation Study on Vertex Pair Closeness

We conduct a necessity analysis of the individual components of Equation 1. We carry out ablation experiments to accurately assess the importance of each constituent element. Specifically, we design three sets of experiments in which only the first term, the second term, and all terms of Equation 1 are used, respectively, to define the closeness between vertex pairs. Subsequently, we have selected the widely-used BFS algorithm to evaluate its acceleration ratio and cache miss rate across eight different datasets for these experimental setups. By comprehensively comparing the performance metrics under these different settings, we quantitatively evaluate the contributions of each component

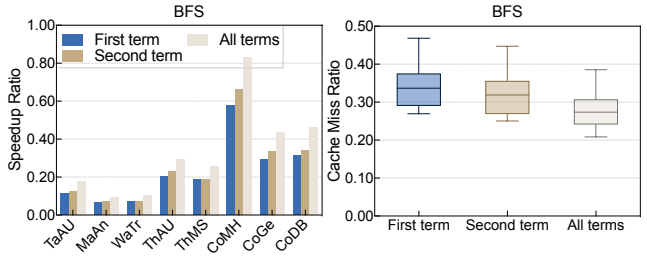


Fig. 6. Ablation study on vertex pair closeness.

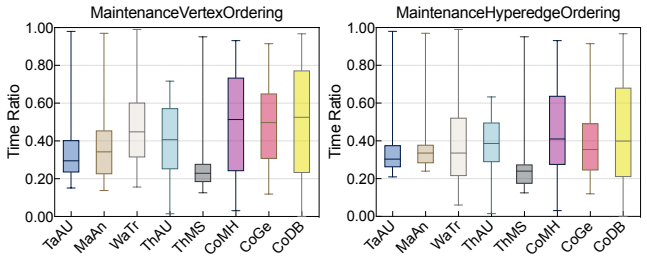


Fig. 7. Time to maintain the algorithm

to the vertex pair closeness. This further provides a robust justification for the design choices made in Equation 1.

In Fig. 6, we delve deeply into the performance differences in vertex pair closeness under the three parameter settings of the first term, the second term, and all terms. As clearly indicated by the speedup ratio on the left side, all terms consistently outperform the other two settings across all datasets. Meanwhile, on some datasets such as CoMH, CoGe, and CoDB, the performance of the second term is notably superior to the first term. This can be attributed to the fact that the second term incorporates neighborhood consistency more comprehensively in the computation of vertex closeness. Further, the cache miss ratio displayed on the right offers more intuitive evidence, showcasing the lower median cache miss ratio of all terms compared to the other two. This validates that all terms effectively combine the advantages of the other two settings, achieving outstanding performance.

6.4 Maintenance Algorithm

Our maintenance algorithms are evaluated on various datasets. For each dataset, we conduct 1000 maintenance experiments and record the ratio of the time of maintenance operations to the time of reordering from scratch for various datasets. The time of maintenance operations and reordering from scratch represent the time required to run the maintenance and reordering algorithms, respectively, when a hyperedge is inserted into or deleted from the hypergraph. The results of both the vertex and hyperedge maintenance algorithms are presented using box plots in Fig. 7.

The results depicted in Fig. 7 illustrate significant variability in the ratio of maintenance time to hypergraph reordering time across different datasets. In most cases, the time required to maintain the algorithm accounts for only 20%-40% of the time required to reorder the hypergraph. However, for some datasets, the maintenance time accounts for 90% of the time required to reorder. Our analysis reveals that the specific ranges requiring updates within the sequence differ for individual hyperedges, depending on the sparsity of the corresponding hypergraph. Notably, for sparse hypergraphs, the maintenance time is considerably lower than the time required for hypergraph reordering. Furthermore, we observe that simultaneous processing of two hyperedges is possible when they have an inclusion relationship with respect to the sequence update. This is due to the implementation of the partial update strategy, which enables efficient processing of large-scale hyperedge updates that include small-scale hyperedge updates.

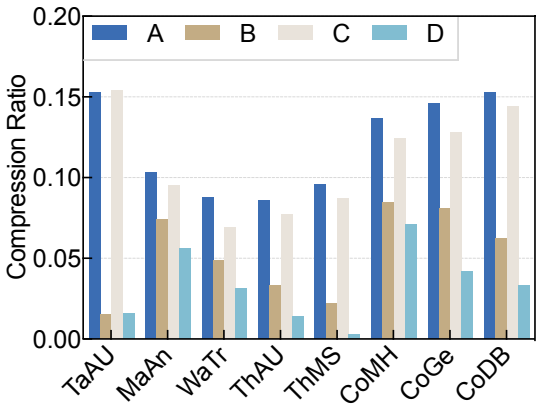


Fig. 8. Compression ratio of our algorithm.

6.5 Compression Evaluation

In this subsection, we assess the efficiency of our algorithm in terms of compression proficiency. We instantiate the elimination of duplicative elements from the hypergraph without affecting the hypergraph processing. To demonstrate the effectiveness of our compression technique, we measure the compression ratio of data structures Array-V&E, Array-V, Array-E, and Array-vanilla relative to Adjlist, as illustrated in Fig. 8, where the compression ratio is (Size of Array-V&E, Array-V, Array-E or Array-vanilla - Size of Adjlist) / (Size of Adjlist).

Figure 8 reveals that the compression ratio of data structures Array-V and Array-E can attain 5%-10%, whereas the compression ratio of data structure Array-V&E can reach 10%-15%. After experimenting with different reordering techniques on hypergraphs, we find that compressing hypergraphs after hyperedge reordering generally produces better results than compressing hypergraphs after vertex reordering. This is because in hypergraphs, the number of identical hyperedges is usually more in hypergraphs. As a result, hyperedge reordering is able to group together more identical hyperedges and compress them effectively, leading to better compression performance. This finding is noteworthy because it contributes to conserving machine memory.

6.6 Integration in Existing Frameworks

We integrate the proposed acceleration structure into Hygra [12], a widely deployed hypergraph processing framework, to verify if our design could benefit existing implementations. We have implemented our reordering algorithm for

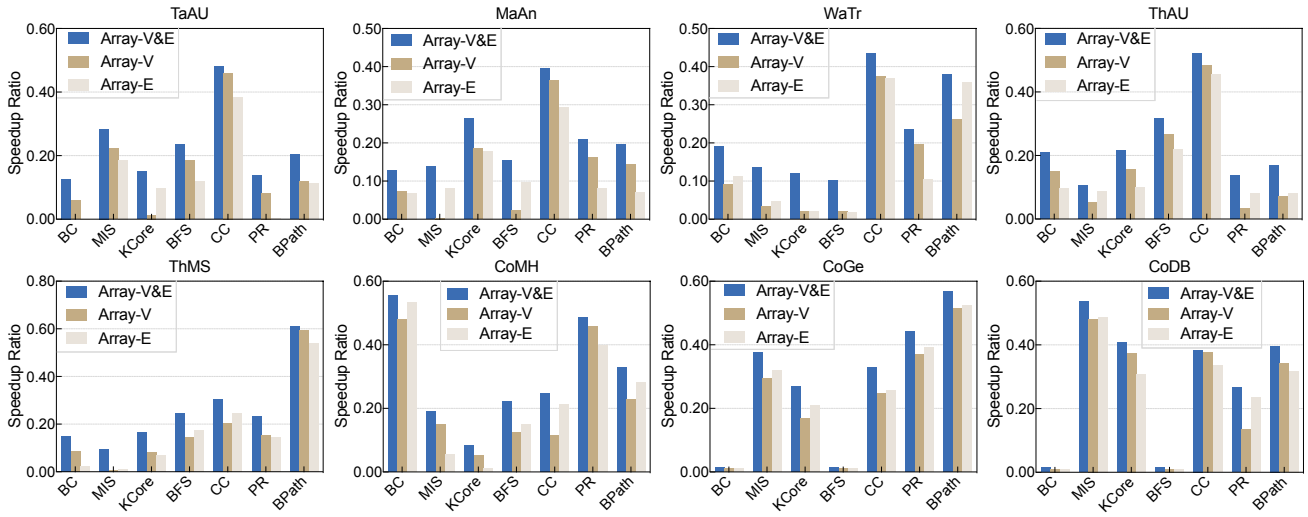


Fig. 9. The Speedup rations of reordering on Hygra.

the Hygra data structure that can reorder the original data in Hygra to enhance data locality and improve data access efficiency. In the experiments, we execute seven hypergraph processing algorithms that Hygra supports, and measure the speedup over the vanilla version. We then record their speedup ratios relative to the original structures, which is shown in Fig. 9.

The experimental results show that integrating our acceleration structure results in speedups ranging from 20% to 60% in most of the datasets with only a limited number of algorithms and datasets yielding suboptimal results in the BC and BFS algorithms of the CoGe and CoDB datasets. After analysis, we find that the original data arrangement of the CoGe and CoDB datasets is more advantageous for BC and BFS algorithms. At the same time, compared to other algorithms such as KCore, PR, etc., BC and BFS cannot better utilize memory locality due to their memory access patterns. As a result, the acceleration effect of the reordering algorithm on BC and BFS is not significant in the CoGe and CoDB datasets. These outcomes on most datasets demonstrate the broad applicability of our reordering algorithm in various algorithms, consistently attaining favorable results.

6.7 Case Study

To demonstrate the advantages of our accelerated structure in a more intuitive and concrete way, we conducted a case study. Our preliminary analysis of the TaAU hyperedge composition reveals that 42.91% of hyperedges contain at least two identical hyperedges, and 17.25% of hyperedges consist of at least 10 identical hyperedges. Consequently, data structure Array-E of hyperedge reordering is necessary to achieve a compression ratio of 15.50%. Moreover, we observe that our methods on TaAU yield better acceleration effects on algorithms such as SCC and PR that apply more hyperedge operators, reaching 22.23% and 26.10%, respectively. This approach is more effective than vertex reordering since TaAU has a higher proportion of repeated hyperedges, thereby allowing the hyperedge reordering algorithm to yield superior results.

7 Conclusion

This paper aims to improve the efficiency of hypergraph processing by enhancing the locality of hypergraphs storage. To achieve this, we introduce a one-dimensional array as a method for storing hypergraphs and propose a greedy algorithm aimed at the removal of redundant components in hypergraph data. Moreover, we devise a reordering algorithm specifically designed for the effective elimination of duplicate elements within the hypergraph structure. Based on the definition of Vertex Pair Closeness, we present an efficient algorithm to compute an optimal vertex sequence in a hypergraph for hypergraph reordering, offering an $\frac{1}{2w}$-approximation ratio. Since hypergraphs are inherently dynamic, we formulate a localized update maintenance algorithm for dynamic hypergraph reordering. Through exhaustive experimentation, our methodology has been demonstrated to not only enhance the performance of hypergraph algorithms, but also to effectively update reordering results without necessitating costly recomputations. Furthermore, our methods can be extended to any hypergraph processing framework directly, such as Hygra.

Our future research endeavors to further optimize the hypergraph reordering algorithm, as well as its extension to more relational data. Meanwhile, we are committed to utilizing multi-core CPUs to accelerate the parallelization of hypergraph processing. We plan to conduct in-depth research on how to promote parallelism in hypergraph processing by optimizing algorithmic logic, improving data structures, and introducing innovative parallel computing models.

References

[1] Q. Luo, D. Yu, H. Sheng, J. Yu, and X. Cheng, "Distributed algorithm for truss maintenance in dynamic graphs," in *International Conference on Parallel and Distributed Computing: Applications and Technologies*. Springer, 2020, pp. 104–115.

[2] Q. Luo, D. Yu, Z. Cai, X. Lin, and X. Cheng, "Hypercore maintenance in dynamic hypergraphs," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 2051–2056.

[3] Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, "Hypergraph neural networks," in *AAAI Conference on Artificial Intelligence*, 2018.

[4] X. Ouvrard, J.-M. L. Goff, and S. Marchand-Maillet, "Networks of collaborations: Hypergraph modeling and visualisation," *ArXiv*, vol. abs/1707.00115, 2017.

[5] E. Y. Ramadan, A. Tarafdar, and A. Pothen, "A hypergraph model for the yeast protein complex network," *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 189–, 2004.

[6] M. Leordeanu and C. Sminchisescu, "Efficient hypergraph clustering," in *Artificial Intelligence and Statistics*. PMLR, 2012, pp. 676–684.

[7] P. Valdivia, P. Buono, C. Plaisant, N. Dufournaud, and J.-D. Fekete, "Analyzing dynamic hypergraphs with parallel aggregated ordered hypergraph visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, pp. 1–13, 2021.

[8] S. Maleki, U. K. Agarwal, M. Burtscher, and K. Pingali, "Bipart: a parallel and deterministic hypergraph partitioner," *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020.

[9] D. Cai, M. Song, C. Sun, B. Zhang, linda Qiao, and H. Li, "Hypergraph structure learning for hypergraph neural networks," in *IJCAI*, 2022.

[10] C. Zhu, Q. Liu, and J. Bi, "Communication efficient distributed hypergraph clustering," *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021.

[11] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1813–1828.

[12] J. Shun, "Practical parallel hypergraph algorithms," *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020.

[13] A. J. Smith, "Line (block) size choice for cpu cache memories," *IEEE transactions on computers*, vol. 100, no. 9, pp. 1063–1075, 1987.

[14] J.-K. Peir, W. W. Hsu, and A. J. Smith, "Functional implementation techniques for cpu cache memories," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 100–110, 1999.

[15] J. Fang, Q. Fan, X. Hao, Y. Cheng, and L. Sun, "Performance optimization by dynamically altering cache replacement algorithm in cpu-gpu heterogeneous multi-core architecture," *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 723–726, 2017.

[16] J. Tse and A. J. Smith, "Cpu cache prefetching: Timing evaluation of hardware implementations," *IEEE Transactions on Computers*, vol. 47, no. 5, pp. 509–526, 1998.

[17] Y.-Y. Jo, M.-H. Jang, S.-W. Kim, and S. Park, "A data layout with good data locality for single-machine based graph engines," *IEEE Transactions on Computers*, vol. 71, pp. 1784–1793, 2021.

[18] P. Gao, M. Zhang, K. Chen, Y. Wu, and W. Zheng, "High performance graph processing with locality oriented design," *IEEE Transactions on Computers*, vol. 66, pp. 1261–1267, 2017.

[19] W. Chen, I. Paik, and Z. Li, "Tology-aware optimal data placement algorithm for network traffic optimization," *IEEE Transactions on Computers*, vol. 65, pp. 2603–2617, 2016.

[20] Y. Hua, X. Liu, and D. Feng, "Data similarity-aware computation infrastructure for the cloud," *IEEE Transactions on Computers*, vol. 63, pp. 3–16, 2014.

[21] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. D. Nguyen, Y. kuang Chen, and P. K. Dubey, "Cache-conscious frequent pattern mining on a modern processor," in *Very Large Data Bases Conference*, 2005.

[22] A. K. Datta and R. Patel, "Cpu scheduling for power/energy management on multicore processors using cache miss and context switch data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 1190–1199, 2014.

[23] S. Niu and S. Chen, "Optimizing cpu cache performance for pregel-like graph computation," in *2015 31st IEEE International Conference on Data Engineering Workshops*. IEEE, 2015, pp. 149–154.

[24] S. Noll, J. Teubner, N. May, and A. Böhm, "Accelerating concurrent workloads with cpu cache partitioning," *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 437–448, 2018.

[25] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu, "Random walks on huge graphs at cache efficiency," *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[26] K. Lakhotia, R. Kannan, S. Pati, and V. K. Prasanna, "Gpop: A scalable cache- and memory-efficient framework for graph processing over parts," *ACM Trans. Parallel Comput.*, vol. 7, pp. 7:1–7:24, 2020.

[27] J.-S. Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 769–782, 2002.

[28] A. Mukkara, N. Beckmann, and D. Sanchez, "Cache-guided scheduling: Exploiting caches to maximize locality in graph processing," *AGP'17*, 2017.

[29] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[30] J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen, "Graphm: an efficient storage system for high throughput of concurrent graph processing," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[31] E. Lee, J. Kim, K. Lim, S. H. Noh, and J. Seo, "Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines," in *USENIX Annual Technical Conference*, 2019.

[32] A. R. Benson, R. Abebe, M. T. Schaub, A. Jadbabaie, and J. Kleinberg, "Simplicial closure and higher-order link prediction," *Proceedings of the National Academy of Sciences*, 2018.

[33] N. Veldt, A. R. Benson, and J. Kleinberg, "Minimizing localized ratio cut objectives in hypergraphs," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2020.

[34] I. Amburg, N. Veldt, and A. R. Benson, "Clustering in graphs and hypergraphs with categorical edge labels," in *Proceedings of the Web Conference*, 2020.

[35] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. P. Hsu, and K. Wang, "An overview of microsoft academic service (MAS) and applications," in *Proceedings of the 24th International Conference on World Wide Web*. ACM Press, 2015.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[37] X.-S. Yang, "Introduction to algorithms," *Nature-Inspired Optimization Algorithms*, 2021.

[38] F. Kuhn and C. Zheng, "Efficient distributed computation of mis and generalized mis in linear hypergraphs," *ArXiv*, vol. abs/1805.03357, 2018.

[39] K. Ochs, D. Michaelis, and E. Solan, "Solving the longest path problem using a hfo2-based wave digital memristor model," in *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2019, pp. 355–358.

Yu Liu received the B.S. degree in Department of Computer Science and Technology from Shandong University, China, in 2021. He is currently working toward the Ph.D degree with the Department of Computer Science and Technology, Shandong University. His research interests include dynamic graphs and parallel computing.

Qi Luo received Ph.D degree from the Department of Computer Science and Technology, Shandong University in 2022. His research interests include graph analytics and distributed computing.

Mengbai Xiao is currently a Qilu Young Professor in School of Computer Science and Technology at Shandong University. Before this, he was a postdoctoral researcher in Department of Computer Science and Engineering at the Ohio State University for two years. He received my Ph.D. from the Department of Computer Science at George Mason University in 2018.

Dongxiao Yu received the B.S. degree in 2006 from the School of Mathematics, Shandong University and the Ph.D degree in 2014 from the Department of Computer Science, The University of Hong Kong. He is currently a professor in the School of Computer Science and Technology, Shandong University. His research interests include wireless networks, distributed computing and graph algorithms.

Huashan Chen received the B.S. and M.Sc. degrees in Computer Science from Shandong University, and the Institute of Information Engineering, Chinese Academy of Sciences, respectively. In December 2021, he received the Ph.D. degree in Computer Science at the University of Texas at San Antonio. He joined the Institute of Information Engineering as a Young Associate Professor in 2022.

Xiuzhen Cheng is an Assistant Professor in the Department of Computer Science at the George Washington University. She received her MS and PhD degrees in Computer Science from the University of Minnesota - Twin Cities in 2000 and 2002, respectively. Her current research interests include Wireless and Mobile Computing, Sensor Networks, Wireless Security, Statistical Pattern Recognition, Approximation Algorithm Design and Analysis, and Computational Medicine. She is an editor for the International Journal on Ad Hoc and Ubiquitous Computing and the International Journal of Sensor Networks.

APPENDIX

Theorem 1. The time complexity of Algorithm 1 is $O(n * \overline{d} + m * \overline{c})$ and its space complexity is $O(\overline{c})$.

Proof: The time complexity of Algorithm 1 is dependent on detecting duplicate parts within two sets (Line 10). To solve this problem, we use a hash array for quick deduplication. First, we hash one set and then verify whether the vertices in the other set exist in the hash list, which takes $O(1)$ time for each vertex. This operation takes $O(\overline{c})$ time (Lines 5-17). Thus, the total time for all vertices is $O(m * \overline{c})$ (Lines 4-17). Similarly, for all incident hyperedges, it takes $O(n * \overline{d})$. Combining all of the above, Algorithm 1 takes $O(n * \overline{d} + m * \overline{c})$ time to compress a hypergraph.

The space complexity of Algorithm 1 is mainly caused by the hash array (Line 10). Thus, we place emphasis on analyzing the size of this hash array. Since Line 10 aims to identify the intersection of two sets, the size of the hash array corresponds to the size of the hyperedge. Consequently, the space complexity of Algorithm 1 is $O(\overline{c})$. □

Theorem 2. Given a hypergraph $G = (V, E)$, finding an optimal sequence $\mathcal{L}$ to maximize $F(\mathcal{L})$ is NP-hard.

Proof: Considering a special case $w = 2$, Equation 2 is written as

$$F(\mathcal{L}) = \sum_{i=1}^{n-1} S(u_i, v_{i+1}) \tag{7}$$

To compute Equation 7, we construct a pairwise graph $G' = (V, E)$, where $V(G') = V(G)$ and $E(G') = V(G') * V(G')$. In G', we assign each edge $e = (u, v) \in E(G')$ a weight of $S(u, v)$, which is computed in the original hypergraph G. This approach enables us to convert the computation of Equation 7 into a search for the longest path in the pairwise graph G' that traverses each vertex only once. This search is equivalent to the longest path problem, which has been proven to be NP-hard [39]. As Problem 1 is no simpler than the longest path problem, we demonstrate that the problem of finding an optimal sequence $\mathcal{L}$ to maximize $F(\mathcal{L})$ is NP-hard. □

Theorem 3. For a hypergraph $G = (V, E)$, the time complexity of Algorithm 2 is $O(\sum_{u \in V}(mdeg(u))^2 + n^2)$.

Proof: The time complexity of Algorithm 2 can be divided into two parts: the update of $FV(\cdot)$ (Line 9-22) and identifying a vertex with the maximum $FV(\cdot)$ (Line 23).

Initially, let's consider the update of $FV(\cdot)$. For each vertex $u \in V$, Algorithm 2 updates the associated $FV(\cdot)$ only once (Line 9-22). Using the definition of $mdeg(\cdot)$, Line 12-14 is executed $mdeg(u)$ times. Therefore, the update of $FV(\cdot)$ for vertex u is executed $\sum_{v \in e, e \in N_G(u)} mdeg(v)$ times. In total, the update of $FV(\cdot)$ is executed $\sum_{u \in V} \sum_{v \in e, e \in N_G(u)} mdeg(v)$ times for the entire algorithm, which simplifies to $\sum_{u \in V}(mdeg(u))^2$. Hence, the time complexity of the update of $FV(\cdot)$ is $O(\sum_{u \in V}(mdeg(u))^2)$.

Next, we consider the time complexity of finding a vertex with the maximum $FV(\cdot)$ (Line 23). Algorithm 2 finds the vertex with the maximum $FV(\cdot)$ n times, and each time the complexity is $|V_c|$. Therefore, the time complexity of finding the vertex with the maximum $FV(\cdot)$ is $O(\sum_{i=1}^{n} |V_{c_i}|) = O(\sum_{i=1}^{n}(n - i)) = o(n^2)$.

In summary, the time complexity of Algorithm 2 is $O(\sum_{u\in V}(mdeg(u))^2 + n^2)$. □

Theorem 4. Algorithm 2 give $\frac{1}{2w}$-approximation for finding an optimal sequence $\mathcal{L}$ that maximizes $F(\mathcal{L})$.

Proof: Let $\overline{F}(\mathcal{L})$ as the score of the optimal solution of Problem 1. And we denote $\hat{F}(\mathcal{L})$ as the score of the ordering result by the Algorithm 2. Then, we give the form of linear programming for the optimal solution of Problem 1.

$$\begin{aligned} \overline{F}(\mathcal{L}) = \max & \sum_{i=1}^{n-1}\sum_{i+1} s_{ij} * x_{ij} \\ \text{s.t.} & \sum_{j>x} x_{ij} + \sum_{j<i} x_{ji} = 2w, i, j \in [1, n] \\ & 0 \le x_{ij} \le 1, \qquad\qquad i, j \in [1, n] \end{aligned} \tag{8}$$

where s_{ij} represents $S(u_i, v_j)$ of the pair of vertices u_i, v_j in the original hypergraph. According to Equation 8, if v_i and v_j fall within a window of width w, the relevant constraints can be satisfied by setting $x_{ij} = 1$. Otherwise, $x_{ij} = 0$. For any $i \in [1, n]$, it requires $\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2w$ according to Equation 8. Consider the node v_i, let W_i be the set containing the w preceding nodes and the w succeeding nodes of v_i, along with v_i itself, such that $|W_i| = 2w + 1$. Furthermore, we formulate $\overline{F}(\mathcal{L})$ by the Lagrangian duality.

$$\begin{aligned} \overline{F}(\mathcal{L}) &= \max \sum_{i=1}^{n-1}\sum_{i+1} s_{ij} * x_{ij} + \sum_{i=1}^{n} \alpha_i (2w - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji}) \\ &= \max \sum_{i=1}^{n-1}\sum_{i+1} (s_{ij} - \alpha_i - \alpha_j) * x_{ij} + 2w \sum_{i=1}^{n} \alpha_i \end{aligned} \tag{9}$$

where α_i is Lagrange multiplier. Assume the hypergraph ordering is $u_1, u_2, \ldots, u_n$, where $\delta(u_i) = i$. In Equation 9, α_i is the maximum sum of the closeness of u_i and the $w-1$ vertices before u_i in the optimal sequence, i.e., $\alpha_i = FV(\mathcal{L}[i])$. It holds that $\alpha_i \ge 0$ and $\sum_i^n \alpha_i = \hat{F}(\mathcal{L})$. According to the greedy algorithm, $s_{ij} - \alpha_i \le 0$, it indicates that $s_{ij} - \alpha_i - \alpha_j \le 0$. Therefore, we have $\overline{F} \le 2w \sum_i^n \alpha_i = 2w * \hat{F}(\mathcal{L})$. Consequently, it follows that $\hat{F}(\mathcal{L}) \ge \frac{1}{2w}\overline{F}(\mathcal{L})$, which results in the $\frac{1}{2w}$-approximation. □

Theorem 5. Given a hypergraph $G = (V, E)$ and a hyperedge e, when the hyperedge e is inserted into or deleted from the hypergraph G, $FV(u)$ changes only if $u \in \phi(e)$, for any vertex u.

Proof: To establish the validity of Theorem 5, it is essential to initially fathom the closeness composition of a vertex within a specified sequence. Equation 4 reveals that the closeness of any vertex v depends upon the $w-1$ vertices preceding it. Moreover, Equation 1 exhibits that the closeness of any pair of vertices $S(u, v)$ is contingent on two constituents: $S_c(u, v)$ and $S_p(u, v)$. We analyze both cases separately, wherein the vertex u either belongs to the set $\phi(e)$ or does not.

Case 1 ($u \in \phi(e)$): We divide this case into two subcases based on whether the vertex u belongs to the hyperedge e or not.

1) ($u \in e$, and $\forall v \in e$): If the hyperedge e is inserted into or deleted from the hypergraph G, it leads to an obvious change in the value of $S_c(u, v)$, given that both vertices u and v belong to the hyperedge e.
2) ($u \notin e$, and $\forall v \in e$): By the definition of hyperedge-neighbor $\phi(e)$, vertex u can access vertex v through a path $p(u, v_2)$ containing the hyperedge e. Consequently, any insertion or deletion of the hyperedge e leads to a modification in the value of $S_p(u, v_2)$.

Combining the aforementioned analyses, we conclude that the values of $S(u, v)$ for any given pair of vertices (u, v) undergo a modification upon the insertion or deletion of the hyperedge e. Furthermore, as per Equation 4, $FV(u)$ may change if the vertex v belongs to the $w-1$ vertices preceding u.

Case 2 ($v \notin \phi(e)$): In the scenario where v does not belong to the hyperedge-neighbor set $\phi(e)$, it follows that v is not a part of the hyperedge e, and hence the incident hyperedges of v remain unaltered. As a result, the value of $S_c(v, \cdot)$ remains unchanged. Furthermore, we deduce from the definition of hyperedge-neighbor $\phi(e)$ that a vertex v cannot access any vertex within $\phi(e)$ through a path $p(v, \cdot)$ containing the hyperedge e. Therefore, there is no change in the value of $S_p(v, \cdot)$.

Combining all of the above, We can establish the validity of Theorem 5. □

Theorem 6. When a hyperedge e is either inserted into or deleted from the hypergraph $G = (V, E)$, it follows that any vertex located within the sequence $\mathcal{L}[lb, ub]$ is susceptible to modifications. The sequence $\mathcal{L}$ represents the original sequence, while $\mathcal{L}'$ denotes the updated sequence. Subsequently, we define lb and ub as follows:

1) $lb = \min\{\delta(u) | u \in \phi(e)\} + 1$.
2) $\mathcal{L}[lb, ub]$ contains all the vertices in $\phi(e)$ and $\mathcal{L}[ub - w + 1, ub] = \mathcal{L}'[ub - w + 1, ub]$.

Proof: As Equation 4 illustrates, the closeness of any given vertex v is dependent upon the $w-1$ vertices that precede it. Thus, we can deduce that when a hyperedge is inserted, there is no alteration to $\mathcal{L}[lb]$ as the feature array $FV(\cdot)$ of the preceding $w-1$ vertices remains unmodified, as per Theorem 5. Consequently, the initial position at which vertex updates may occur is lb. Moreover, if the sequences $\mathcal{L}$ and $\mathcal{L}'$ have the same segment of length W, it is evident that no changes occur for the vertices following the position ub. This is because the feature array $FV(\cdot)$ of each vertex is solely dependent on the $w-1$ vertices that precede it, as per Theorem 5.

Taking into account the aforementioned analyses, we can conclude that only the vertices within the segment $\mathcal{L}[lb, ub]$ are susceptible to modifications. □

Theorem 7. Given a hypergraph $G = (V, E)$ and a hyperedge e, the worst-case time complexity of Algorithm 3 is $O(\sum_{u\in V}(mdeg(u))^2 + n^2)$.

Proof: Algorithm 3 and Algorithm 2 utilize the same update strategy, where the vertex with the maximum $FV(\cdot)$ value is selected to be added to the sequence. Thus, in the worst-case scenario where all vertices in the sequence require an update, the time complexity of Algorithm 3 is consistent with that of Algorithm 2, that is $O(\sum_{u\in V}(mdeg(u))^2 + n^2)$. □