OHMiner: An Overlap-centric System for Efficient Hypergraph Pattern Mining

Hao Qi[1], Kang Luo[1], Ligang He[2], Yu Zhang[1], Minzhi Cai[1], Jingxin Dai[1], Bingsheng He[3], Hai Jin[1], Zhan Zhang[4], Jin Zhao[1], Hengshan Yue[5], Hui Yu[1], Xiaofei Liao[1]

[1] National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China

[2] Department of Computer Science, University of Warwick, United Kingdom

[3] National University of Singapore, Singapore [4] Zhejiang Lab, China [5] Jilin University, China

{theqihao,luokang2000,zhyu,caimz,daijingxin,hjin,zjin,huiy,xfliao}@hust.edu.cn,ligang.he@warwick.ac.uk, dcsheb@nus.edu.sg,kira789632147@gmail.com,yuehs@jlu.edu.cn

Abstract

Hypergraph Pattern Mining (HPM) aims to identify all the instances of user-interested subhypergraphs (patterns) in hypergraphs, which has been widely used in various applications. However, existing solutions either need significant enumeration overhead because they extend subhypergraphs at the granularity of vertices, or suffer from massive redundant computations because they often need to repeatedly fetch and process the same incident hyperedges for different vertices. This paper presents an *overlap-centric* system named OHMiner to efficiently support HPM. OHMiner proposes an overlap-centric execution model to determine the subhypergraphs isomorphism through computing and comparing overlaps among hyperedges using set operations. This model aims to efficiently handle the vertices that collectively share the same incident hyperedges. To automatically and precisely retrieve an arbitrary pattern's overlapping semantics without performing redundant set computations, OHMiner further proposes a redundancy-free compiler, which constructs an *Overlap Intersection Graph* (OIG) for the pattern, optimizes the OIG, and generates an overlap-centric execution plan to guide the procedure of HPM. Moreover, OHMiner designs an overlap-centric parallel execution engine, which adopts an incremental overlap-pruned approach to fast validate candidates for HPM. Additionally, it proposes a degree-aware data store to support efficient generation of candidates. Through evaluating OHMiner on a broad range of real-world hypergraphs with various patterns, our experimental results show that OHMiner outperforms the state-of-the-art HPM system by 5.4×–22.2×.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; • **Mathematics of computing → Graph theory**; • **Software and its engineering → Compilers**.

Keywords: Hypergraph pattern mining, Overlap, Compiler, Parallel execution engine

ACM Reference Format:

Hao Qi, Kang Luo, Ligang He, Yu Zhang, Minzhi Cai, Jingxin Dai, Bingsheng He, Hai Jin, Zhan Zhang, Jin Zhao, Hengshan Yue, Hui Yu, Xiaofei Liao. 2025. OHMiner: An Overlap-centric System for Efficient Hypergraph Pattern Mining. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3717474

1 Introduction

Hypergraph [4, 58], where a hyperedge can contain an arbitrary number of vertices, is a generalization graph model beyond ordinary graphs, in which an edge only represents a pairwise relationship. Hypergraphs can naturally and flexibly describe high-order interactions among multiple entities [22, 29, 40, 49, 64] compared to ordinary graphs. *Hypergraph Pattern Mining* (HPM), also called subhypergraph matching, is one of the most fundamental and significant problems in hypergraphs and involves extracting structural information from hypergraphs. Specifically, HPM aims to explore all subhypergraphs (embeddings), which are isomorphic to a user-specified subhypergraph (pattern), in data hypergraphs. It has been widely used across numerous applications, including specific protein/gene discovery [22, 29, 37, 49, 59], pattern search in collaborative/social networks [27, 39, 66], object detection [13, 34, 41], and machine learning [28, 38, 45]. For instance, hypergraphs can

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3717474

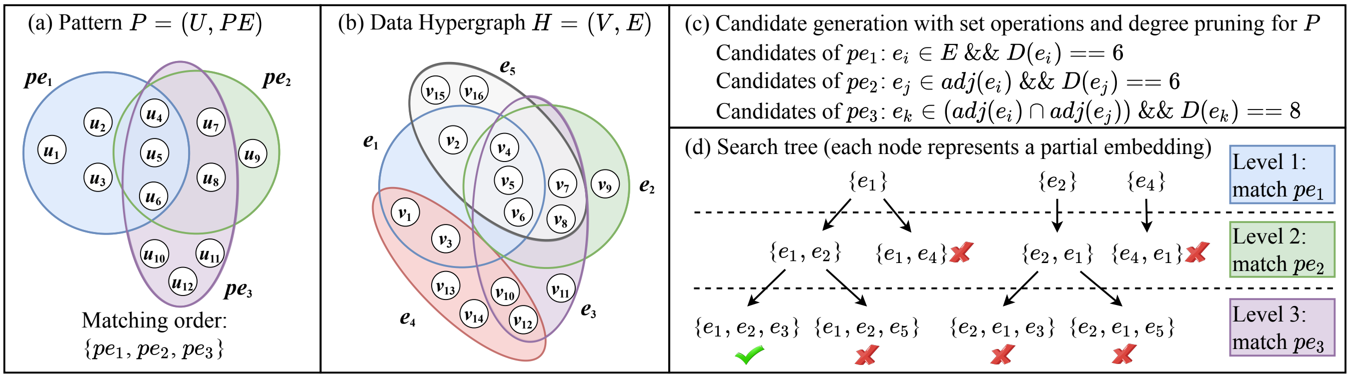


Figure 1. An example to demonstrate procedure of the match-by-hyperedge approach for HPM systems

precisely model protein interactions in protein complex networks where proteins are vertices and protein complexes are hyperedges [29, 49]. This modeling allows biologists to describe protein complexes groups as patterns and search for these patterns in massive biological networks, thereby gaining insights into their interactions and functions.

Similar to subgraph isomorphism in ordinary *Graph Pattern Mining* (GPM) systems [18, 31, 44, 51], subhypergraph isomorphism is a core component to support downstream HPM applications via APIs. However, efficiently executing HPM applications faces significant challenges. First, subhypergraph isomorphism in HPM is an NP problem [2, 3, 11, 42, 46] and thus suffers from high algorithmic complexity. Second, **complexity relationships among hyperedges in HPM make it more difficult to validate subhypergraph isomorphism than subgraph isomorphism in GPM.** This is because the overlap between two hyperedges can contain multiple vertices and can also be contained by multiple hyperedges, while that between two ordinary edges is either null or a single vertex (an overlap is the set of common vertices among distinct hyperedges). Therefore, HPM systems require additional verification steps compared with GPM systems [65]. Despite previous research efforts, existing HPM systems still suffer from high enumeration costs or redundant computation overhead because they support HPM at the vertex-granularity.

Existing systems can be categorized into two approaches: the match-by-vertex approach and the match-by-hyperege approach. Early HPM systems [14, 15, 24, 55, 66] expand GPM to HPM by using a match-by-vertex approach, where it extends subhypergraphs by adding a vertex at a time. This approach enumerates each vertex in the pattern to find its candidates in data hypergraphs and then validates hyperedges, resulting in large search space and significant enumeration overhead. To reduce these issues, HGMatch [65] proposes a match-by-hyperedge approach, which extends subhypergraphs by adding one hyperedge at a time. It generates hyperedge candidates and then validates hyperedge candidates after each extension. However, candidate generation and validation in HGMatch also use the vertex-granularity to retrieve and process vertices' *incident hyperedges* (i.e., hyperedges that contain the vertex). This leads to massive redundant computations (up to 90% of the total time) because multiple vertices can share the same incident hyperedges.

In this work, we find that HPM exhibits significant ***overlap similarity***, meaning that multiple vertices in an overlap among hyperedges share the same incident hyperedges. Similarly, multiple vertices in a hyperedge beyond the overlaps also share the same incident hyperedges. Furthermore, multiple vertices in an overlap among hyperedges beyond other smaller-scope overlaps also share the same incident hyperedges. We consider these vertices redundant. For example, in Figure 1(a), u_4, u_5, and u_6 are vertices in the overlap among hyperedges $\{pe_1, pe_2, pe_3\}$, indicating that they can be handled together. We counted the redundant vertices in the bottleneck procedure (i.e., candidate validation) of HGMatch and found that they accounted for 68%–91% of all vertices. This means that a majority proportion of the total execution time is wasted for candidate validation in HGMatch.

We further find that these vertices can be modeled as a region in the Venn diagram [50], which represents relationships between multiple sets (each set in a Venn diagram corresponds to a hyperedge). Based on this finding, we propose an **overlap-centric execution model** to validate subhypergraph isomorphism by computing and comparing the region size of the Venn diagram that models the hypergraph between the pattern and embedding. Using this model, we find that previous results of repeated set intersection computations can be reused. To further optimize result reuses, we leverage the *inclusion-exclusion principle* to convert set differences into set intersections.

However, there are two main challenges to implement the overlap-centric execution model and its optimization opportunities into an efficient and generic HPM system. First, the real-world hyperedge patterns are complex and diverse. Manually producing HPM solutions to analyze an arbitrary pattern is tedious and time-consuming for programmers and users. Second, maintaining system efficiency and correctness is difficult. The overlapping semantics are derived from pattern analysis. In the HPM, we need to efficiently and correctly match overlaps of partial embeddings from hypergraphs with the corresponding hyperedges.

To overcome these challenges, we present an overlap-centric system called OHMiner , which includes a redundancy-free compiler, an overlap-centric parallel execution engine, and a degree-aware data store. The compiler automatically analyzes an arbitrary pattern by constructing and optimizing an *Overlap Intersection Graph* (OIG), and then generates an overlap-centric execution plan to guide the procedure of HPM. Based on the execution plan, the engine adopts an incremental overlap-pruned approach to validating candidates by computing overlaps and pruning false partial embeddings (subhypergraphs) efficiently. Moreover, the degree-aware data store can support fast candidate generation by quickly determining connection and disconnection relationships among hyperedges. We evaluate OHMiner on a broad range of real-world hypergraphs with various patterns. The results demonstrate that OHMiner outperforms the cutting-edge system HGMatch by 5.4×–22.2×.

In summary, this paper makes the following contributions:

- We propose an overlap-centric execution model to efficiently validate subhypergraph isomorphism.
- We introduce a redundancy-free compiler that automatically analyzes the overlapping semantics of arbitrary patterns to eliminate redundant set computations and generates an overlap-centric execution plan to guide the overlap-centric HPM.
- We present an incremental overlap-pruned approach and a degree-aware data store to efficiently validate candidates and generate candidates, respectively.
- We develop and evaluate a prototype of OHMiner and the results verify the efficiency of OHMiner.

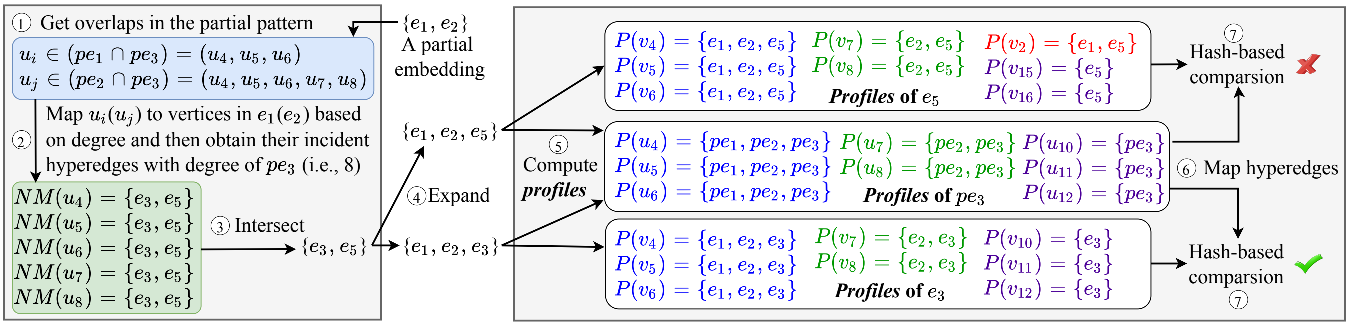
2 Background and Motivation

2.1 Definitions

- **Hypergraph.** A hypergraph H is defined as $H = (V,\ E,\ L)$, where V is a set of vertices, E is a set of hyperedges, each of which is a non-empty subset of V (note that each edge of an ordinary graph has only two vertices), and L is a labeling function that assigns a label to each vertex. $|V|$, $|E|$, and $|L|$ denote the number of vertices, hyperedges, and labels of vertices in H, respectively. For illustration, Figure 1(b) shows a hypergraph with five hyperedges $\{e_1, e_2, e_3, e_4, e_5\}$ and sixteen vertices $\{v_1, v_2, \dots, v_{16}\}$. Note that for convenience of presentation, we use unlabeled hypergraphs in this paper, but our system supports both labeled and unlabeled hypergraphs (see Section 4.3.1).
- **Incident Hyperedges/Vertices.** We say a vertex v and a hyperedge e are *incident* if $v \in e$ $(v \in V,\ e \in E)$. We use $N(e)$ to represent the e's incident vertices and $N(v)$ to represent the v's incident hyperedges (e is also called v's incident hyperedge).
- **Degree of Hyperedge/Vertex.** The number of vertices in $N(e)$ or the number of hyperedges in $N(v)$ is called the *degree* of e or v, denoted as $D(e)$ or $D(v)$. For example, the degree of e_1 is 6, i.e., $D(e_1) = 6$.
- **Overlap.** We say two hyperedges (e_i and e_j) overlap (are connected) when they contain at least one shared vertex (i.e., the **overlap** is $N(e_i) \cap N(e_j) \neq \emptyset$). For example, the overlap between e_1 and e_2 is $\{v_4, v_5, v_6\}$.
- **Adjacency List of Hyperedge.** We represent the list of hyperedges that overlap with a hyperedge e_i as $adj(e_i)$ (or $A(e_i)$). For example, $adj(e_4)$ is $\{e_1, e_3\}$.
- **Subhypergraph.** A subhypergraph $SH = (SV,\ SE)$ is defined as a subset of vertices and hyperedges of H.
- ***Hypergraph Pattern Mining* (HPM).** Given a pattern hypergraph P and a data hypergraph H, HPM (also called subhypergraph matching) aims to find all subhypergraphs (embeddings) that are *isomorphic* to P in H. We say a subhypergraph sh in H is subhypergraph isomorphic to P if and only if there exists a *bijective* mapping of the vertices between sh and P, which induces a bijection of their hyperedges. Specifically, the mapping function is $f : V(P) \rightarrow V(sh)$ such that $\forall e_p = \{u_1, u_2, \dots, u_n\} \subseteq V(P) : e_p \in E(P) \iff e_{sh} = \{f(u_1), f(u_2), \dots, f(u_n)\} \in E(sh)$ [12].
- **Matching Order.** We define the matching order of a pattern as the total order of the pattern's hyperedges.
- **Pattern and Embedding.** We utilize hyperedge sequences to describe a pattern P and an embedding m, i.e., $\{pe_1, pe_2, \dots, pe_n\}$ and $\{e_1, e_2, \dots, e_n\}$, where $e_i = \{f(u) : u \in pe_i\}$ $(1 \leq i \leq n)$. A partial pattern p' contains the first x $(1 \leq x \leq n)$ hyperedges of P, and its corresponding embedding is a partial embedding m'. Figure 1 shows that the pattern's matching order is $\{pe_1, pe_2, pe_3\}$, and a corresponding embedding is $\{e_1, e_2, e_3\}$ which contains partial embeddings $\{e_1\}$ and $\{e_1, e_2\}$.
- **Candidates of Hyperedge.** It is a hypergraph's hyperedges mapped from a pattern's hyperedge (e.g., pe_i) based on connection relationships among hyperedges in partial embeddings and the degree of pe_i. We refer to a candidate of pe_i as c_i. As illustrated in Figure 1(d), given a partial embedding $\{e_1\}$, candidates of pe_2 are $\{e_2, e_4\}$ because both e_2 and e_4 are connected to e_1 and their degree is 6.

2.2 Different from *Graph Pattern Mining* (GPM)

HPM is more challenging than GPM since hypergraphs have more complex relationships among hyperedges. Specifically, given a matching order of vertices for GPM, the partial embedding isomorphism can be uniquely determined by examining whether the newly matched vertex v_i is connected to the previously matched vertices. This is because the overlap of multiple pairwise edges that involve v_i is a unique vertex (i.e., v_i). In contrast, subhypergraph isomorphism cannot be



(a) Candidate generation (b) Candidate validation

Figure 2. An example to show redundant computations of the match-by-hyperedge approach. $NM(u)$ represents the incident hyperedges of the vertices in the hypergraph mapped by vertex u in the pattern, and the degree of the hyperedge needs to be the same as the degree of the extended hyperedge in the pattern. $P(v)$ represents vertex v's *profile*, which includes vertex v's incident hyperedges in the extended partial embedding.

directly determined by examining if the newly matched hyperedge is connected to previously matched hyperedges because the overlaps between hyperedges are complicated (e.g., each overlap can contain multiple vertices and can be contained by multiple hyperedges). **As a result, a validation mechanism (for verifying how the hyperedges are connected) has to be implemented in HPM systems [65]**. Note that the validation is not an issue in GPM, but it becomes the bottleneck in HPM. Addressing this challenge and implementing the arising optimization opportunities are central focuses of this paper.

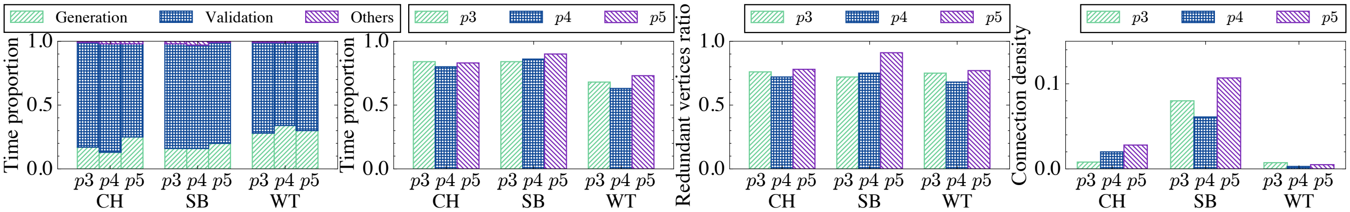
2.3 Problems of the State-of-the-Art HPM Systems

To overcome these challenges, multiple systems have been proposed [14, 15, 24, 55, 65, 66]. In these systems, vertices are the granularity of computation in HPM. These systems can be classified into two categories: match-by-vertex [14, 15, 24, 55, 66] and match-by-hyperedge [65]. The former iteratively maps a pattern vertex to a data vertex in the hypergraph and then validates hyperedges, resulting in large search space and expensive enumeration costs. To reduce these issues, HGMatch [65] proposes a match-by-hyperedge approach and shows significant performance improvement (by four orders of magnitude on average) compared with the previous studies [14, 15, 24, 55, 66]. It extends a subhypergraph by adding a hyperedge and validates the extended subhypergraph for each extension. The process of extending embeddings can be modeled with a search tree (Figure 1(d)), where each node represents a subhypergraph (i.e., a partial embedding), and the subhypergraphs at the level $k + 1$ are extended from the subhypergraphs at the previous level k. Given a matching order $\{pe_1, pe_2, pe_3\}$, an extending process contains candidate generation and validation.

Specifically, to match pe_i in P, HGMatch first generates candidates of pe_i. As shown in Figure 1(c), a candidate of pe_3 is c_3, which is connected to c_1 and c_2 (the candidates of pe_1 and pe_2). This is because pe_3 is connected to pe_1 and pe_2, and c_3's degree is the same as the degree of pe_3 (i.e., 8). Then, HGMatch performs candidate validation by determining if there is an isomorphism between the extended subhypergraphs and the corresponding partial pattern. For example, to extend the subhypergraph $\{e_1, e_2\}$, candidates of pe_3 are $\{e_3, e_5\}$ since they are connected to $\{e_1, e_2\}$ and their degrees are both 6. Note that $\{e_1, e_2, e_5\}$ is not a valid embedding since v_2 in e_5 cannot be mapped to any vertex in P. However, we find that this match-by-hyperedge approach [65] suffers from massive redundant computations because it also uses the vertex-granularity to conduct the generation and validation of candidates, which is discussed next.

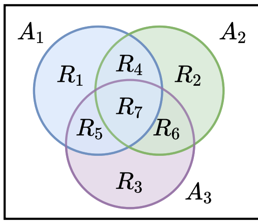
Redundant Computations in Candidate Generation. Figure 2(a) shows the process of extending $\{e_1, e_2\}$ and generating candidates of pe_3. It first traverses all vertices in overlaps between pe_3 and previous hyperedges in P (① in Figure 2), and then maps these vertices to vertices in the corresponding hyperedges of the partial embedding based on vertices' degrees. For example, $u_j \in (pe_2 \cap pe_3)$, and thus u_j is mapped to vertices in e_2. Next, it fetches mapped vertices' incident hyperedges (denoted $NM(u_j)$) in H with the pruning of pe_3's degree (i.e., 8) (②), and intersects these $NM(u_j)$s to generate candidates, i.e., $\{e_3, e_5\}$ (③). However, multiple vertices in an overlap share the same incident hyperedges, leading to redundant computations when computing each vertex's degree and incident hyperedges individually. In Figure 2(a), all $NM(u_j)$ are the same.

Redundant Computations in Candidate Validation. As shown in Figure 2(b), after extending the partial embedding by adding candidates (④), candidate validation is performed to determine the partial subhypergraph isomorphism. To validate a newly added hyperedge candidate e_i, it computes the *profile* of each vertex in e_i (⑤), which includes vertex's incident hyperedges in the extended partial embedding. The profile of v_4 is $P(v_4) = \{e_1, e_2, e_3\}$ since v_4 is contained



(a) Breakdown of HGMatch (b) Redundant computations (c) Redundant vertices of validation (d) Connection density

Figure 3. Statistical studies on the characteristics of HGMatch. pi denotes a pattern with i hyperedges. pi's connection density on the hypergraph H is $C = Cons * 2/(|SE| * (|SE| - 1))$ where H's subhypergraph $SH = (SV, SE)$ are mapped from pi's hyperedges based on their degrees and $Cons$ represents the number of connections between SE.



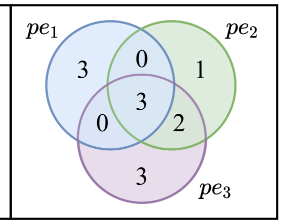
(a) 3-venn diagram

$$\begin{aligned}
R_1 &= A_1 \backslash A_2 \backslash A_3 \\
R_2 &= A_2 \backslash A_1 \backslash A_3 \\
R_3 &= A_3 \backslash A_1 \backslash A_2 \\
R_4 &= A_1 \cap A_2 \backslash A_3 \\
R_5 &= A_1 \cap A_3 \backslash A_2 \\
R_6 &= A_2 \cap A_3 \backslash A_1 \\
R_7 &= A_1 \cap A_2 \cap A_3
\end{aligned}$$

(b) Set operations of R

$$\begin{aligned}
|R_1| &= |A_1| - |A_1 \cap A_2| - |A_1 \cap A_3| + |A_1 \cap A_2 \cap A_3| \\
|R_2| &= |A_2| - |A_1 \cap A_2| - |A_2 \cap A_3| + |A_1 \cap A_2 \cap A_3| \\
|R_3| &= |A_3| - |A_1 \cap A_3| - |A_2 \cap A_3| + |A_1 \cap A_2 \cap A_3| \\
|R_4| &= |A_1 \cap A_2| - |A_1 \cap A_2 \cap A_3| \\
|R_5| &= |A_1 \cap A_3| - |A_1 \cap A_2 \cap A_3| \\
|R_6| &= |A_2 \cap A_3| - |A_1 \cap A_2 \cap A_3| \\
|R_7| &= |A_1 \cap A_2 \cap A_3|
\end{aligned}$$

(c) Set operations of region sizes with IEP



(d) Region sizes of pattern P

Figure 4. An example to motivate our overlap-centric execution model

by hyperedges e_1, e_2, and e_3. Then, it maps the hyperedges in the profiles of the vertices in pe_3 to hyperedges in the partial embedding based on matching order (⑥). For example, $\{pe_1, pe_2, pe_3\}$ are mapped to $\{e_1, e_2, e_3\}$ and $\{e_1, e_2, e_5\}$ for the two extended partial embeddings, respectively. Next, it compares the two profiles through a hash-based method to determine the partial subhypergraph isomorphism (⑦). Note that $\{e_1, e_2, e_5\}$ is not a valid embedding since the profile of v_2 in e_5 cannot equal any profile in the mapped vertices of pe_3, leading to the incorrect hash-based comparison result. However, significant redundancies exist in these profiles, e.g., $P(v_4)$=$P(v_5)$=$P(v_6)$, $P(v_7)$=$P(v_8)$, and $P(v_{10})$=$P(v_{11})$=$P(v_{12})$.

We analyze HGMatch [65] by mining different patterns on several datasets. Section 5 describes the details of the experimental environment, hypergraph datasets, and patterns used in this evaluation. Figure 3 shows the evaluation results of HGMatch. We include only three representative datasets, as the results for the other datasets are similar. Figure 3(a) shows that candidate generation and candidate validation together take up 97% to 99% of the entire time, especially for candidate validation (up to 85%). Figure 3(b) shows that redundant computations in the above procedures account for a significant percentage (up to 90%) of the total time.

3 Overlap-centric Execution Model

To efficiently support the HPM, we propose an overlap-centric execution model, which performs HPM in the granularity of overlap (rather than the traditional granularity of vertex) to collectively handle multiple vertices that share the same incident hyperedges. This model can efficiently determine subhypergraph isomorphism by computing and comparing overlaps among hyperedges.

The overlap-centric execution model stems from our finding that *hypergraph pattern mining exhibits significant* ***overlap similarity****, meaning that multiple vertices in an overlap among hyperedges share the same incident hyperedges. Similarly, multiple vertices in a hyperedge beyond the overlaps also share the same incident hyperedges. Furthermore, multiple vertices in an overlap among hyperedges beyond other smaller-scope overlaps also share the same incident hyperedges.* In Figure 2(b), v_4, v_5, and v_6 are vertices in the overlap among hyperedges $\{e_1, e_2, e_3\}$ and have the same vertex profile, indicating that they can be handled together. Similarly, v_{10}, v_{11}, and v_{12} are the vertices in e_3 that are beyond the overlaps, and they also have the same vertex profile. Figure 3(c) shows that 68%–91% of all vertices are redundant in candidate validation, leading to massive unnecessary and time-consuming computations.

Based on this finding, we propose a hypergraph representation approach, which uses the Venn diagram [50] to model a pattern's vertices, because a region in the Venn diagram can represent the vertices that share the same incident hyperedges. Figure 4(a) shows the 7 regions (excluding the exterior) of the 3-Venn diagram, which models the relationship among three sets (i.e., A_1, A_2, and A_3). A region is denoted by $R_i (1 \leq i \leq 7)$ and can be computed using the set operations in Figure 4(b), where $A_i \setminus A_j$ represents the set difference of A_i and A_j. Consider each set as a hyperedge, vertices in each of the seven regions share the same incident hyperedges and can be processed together. We refer to the

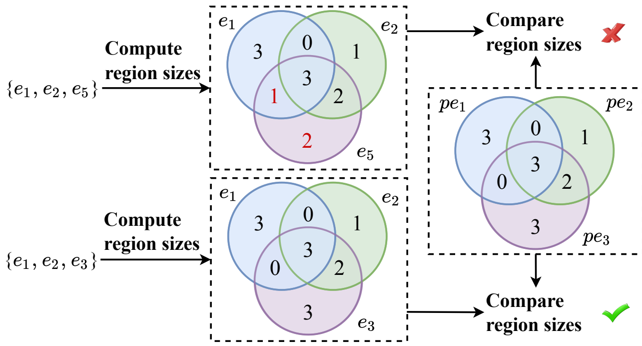


Figure 5. An example to validate two embeddings using their region sizes

number of vertices in a region as the region size. Figure 4(d) shows that the size of regions for the example pattern is {3, 1, 3, 0, 0, 2, 3} for $\{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}$.

After that, we determine subhypergraph isomorphism based on this hypergraph representation. In detail, we first compute the size of each region of the embedding, and then compare it with the corresponding region size of the pattern. If all region sizes are the same, it indicates subhypergraph isomorphism. As shown in Figure 5, $\{e_1, e_2, e_3\}$ is a valid embedding since the size of each region is the same as that of the pattern. The sizes of R_5 and R_3 of $\{e_1, e_2, e_5\}$ are 1 and 2, respectively, which are different from those of the pattern (0 and 3). Therefore, $\{e_1, e_2, e_5\}$ is not a valid embedding. The correctness of our Venn diagram-based approach for subhypergraph isomorphism can be proven by Theorem 1.

Theorem 1. *Given two subhypergraphs, construct a Venn diagram for each subhypergraph. Given an order of hyperedges, whether these two subhypergraphs are isomorphic is equivalent to whether the size of each region in the two Venn diagrams is the same.*

Proof. Each region's incident hyperedges are unique and determined according to the set operations in Figure 4(b). We determine a region order based on the order of hyperedges, and matching region sizes represent that all regions are matched. Each region's vertices share the same incident hyperedges, and consequently every vertex matches (for labeled hypergraphs, we additionally compare the labels of vertices in each region). Therefore, there is a bijection between the two subhypergraphs' vertices, which can induce a bijection between the two subhypergraphs' hyperedges. □

However, we find that **redundant set intersection computations** exist in the set operations. For example, both R_4 and R_7 need to compute $A_1 \cap A_2$. The partial computation result (an overlap) in R_4 can be reused when computing R_7. Set computations for regions involve set differences. To exploit the above advantage (i.e., result reuse) over set differences, we use the *Inclusion-Exclusion Principle* (IEP) [5] to transform set differences to set intersections. Specifically, given m hyperedges, the set operations for each region in the m-Venn diagram can be expressed in the first line of Equation (1), where $1 \le n \le m$. Then, the computation can be further transformed to a set difference between two union operations, each of which can utilize the IEP. Finally, the set differences are completely eliminated. Figure 4(c) shows that the transformed formula for three sets avoids set differences.

$$\begin{aligned}
|R| &= |\bigcap_{i=1}^{n} A_i \setminus A_{n+1} \setminus A_{n+2} \setminus \cdots \setminus A_m| \\
&= |\bigcap_{i=1}^{n} A_i \cup A_{n+1} \cup A_{n+2} \cup \cdots \cup A_m| - |\bigcup_{i=n+1}^{m} A_i| \\
&= |\bigcap_{i=1}^{n} A_i| - \sum_{n+1 \le j \le m} |\bigcap_{i=1}^{n} A_i \cap A_j| + \cdots + (-1)^{m-n} |\bigcap_{i=1}^{m} A_i|
\end{aligned} \tag{1}$$

Utilizing IEP offers two significant benefits. First, the arithmetic operations (i.e., addition and subtraction) involving absolute values (i.e., size of overlaps) in Equation (1) can be completely eliminated once we determine all overlaps among hyperedges. Second, redundant set computations in the Equation can be eliminated. Figure 4(c) shows that $|A_1 \cap A_2|$ and $|A_1 \cap A_2 \cap A_3|$ are repeated 3 times and 7 times, respectively, and $A_1 \cap A_2$ can be used as an intermediate result to compute $A_1 \cap A_2 \cap A_3$. *To this end, we only need to compute overlaps among hyperedges once, which are a part of the set operations for regions, as shown in Figure 4(b).* Note that HGMatch [65] cannot eliminate its redundant computations (see Section 2.3) because it processes HPM at the vertex granularity and does not identify vertices within the same region.

Further Optimization Opportunities for HPM. More importantly, the above approach opens multiple new optimization opportunities for HPM. First, if the sizes of any overlap between a partial pattern and a partial embedding are different, pruning can be performed to avoid unnecessary subsequent set computations. For example, when extending $\{e_1\}$ to $\{e_1, e_4\}$, the computation of $e_1 \cap e_4 \cap e_3$ is unnecessary and can be pruned since $(|pe_1 \cap pe_2| = 3) \ne (|e_1 \cap e_4| = 2)$. Second, not all hyperedges in a pattern are connected to each other, resulting in a significant number of empty overlaps (i.e., the degree of overlap is zero), which can further be used to identify unnecessary computations. For example, if $|A_1 \cap A_2| = 0$, $|A_1 \cap A_2 \cap A_3|$ must be 0 and is an unnecessary computation. Besides, empty overlaps can be determined by checking the disconnections between hyperedges, rather than computing set intersections. Figure 3(d) illustrates that the majority of hyperedges in the data hypergraph's subhypergraph, which is mapped from the pattern's hyperedges based on their degrees, are disconnected (only one tiny fraction, up to 0.11, is connected). This indicates that we can leverage the disconnection relationships among hyperedges to reduce massive set computations for empty overlaps.

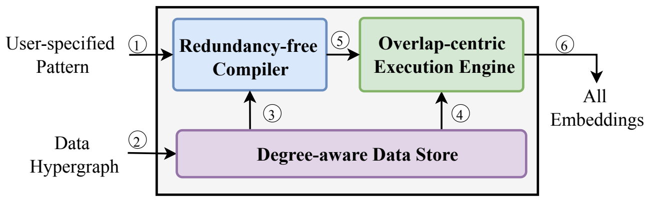


Figure 6. Overview of OHMiner system

4 OHMiner System

4.1 Challenges of System Design

Incorporating the overlap-centric execution model and its optimization opportunities into a general HPM system efficiently faces two key challenges.

- **Complex and Diverse Patterns.** Capturing the overlapping semantics of arbitrary patterns is difficult due to the various complicated relationships between hyperedges. Manually generating HPM solutions to identify and eliminate the aforementioned redundant set computations for arbitrary patterns is tedious and time-consuming for programmers and users. Hence, we propose a compiler to automate pattern analysis.
- **Maintaining System Efficiency and Correctness.** The overlapping semantics are derived from pattern analysis. In the HPM process, we need to efficiently and correctly match overlaps of partial embeddings from hypergraphs with the corresponding hyperedges.

4.2 Overview of OHMiner System

To overcome the above challenges, we present an overlap-centric HPM system OHMiner. Figure 6 shows the overview of OHMiner. OHMiner takes a user-specified pattern as well as a data hypergraph as inputs (① and ② in Figure 6) and then outputs all the embeddings (⑥). Specifically, the **Redundancy-free Compiler** automatically analyzes the pattern (①) and generates an overlap-centric execution plan to guide the execution of HPM (⑤). Given an overlap-centric execution plan (⑤) and hypergraph data (④), the **Overlap-centric Execution Engine** identifies all embeddings in the hypergraph for a pattern (⑥). The **Degree-aware Data Store** first loads data hypergraphs from source files (②) and builds the data structure of the hypergraph for the above two components (③ and ④).

4.3 Redundancy-free Compiler

The overview of the redundancy-free compiler is shown in Figure 7. The front-end of the compiler constructs the high-level intermediate representation, i.e., *Overlap Intersection Graph* (OIG), of an arbitrary pattern and adopts the merge optimization to eliminate redundant set computations. The middle-end analyzes the OIG to generate an order of overlap execution and then utilizes a group-based pruning technique to reduce unnecessary computations of empty overlaps. The

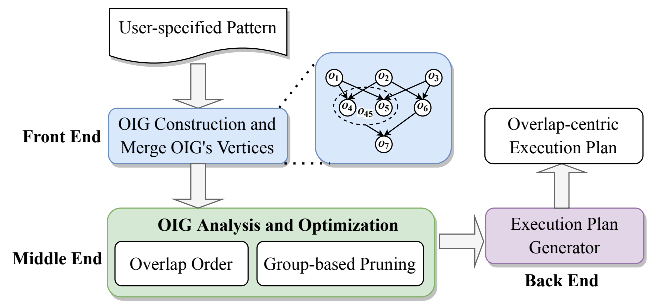


Figure 7. Overview of the redundancy-free compiler

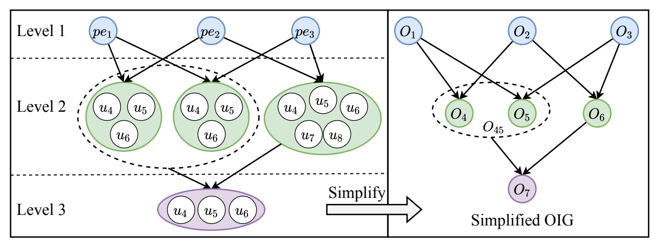


Figure 8. An example of overlap intersection graph (left) and its simplified representation (right)

back-end of the compiler generates the overlap-centric execution plan (low-level intermediate representation) to guide the procedure of HPM.

4.3.1 OIG Construction.

To extract the overlapping semantics of a pattern, we propose an OIG to group vertices with the same incident hyperedges.

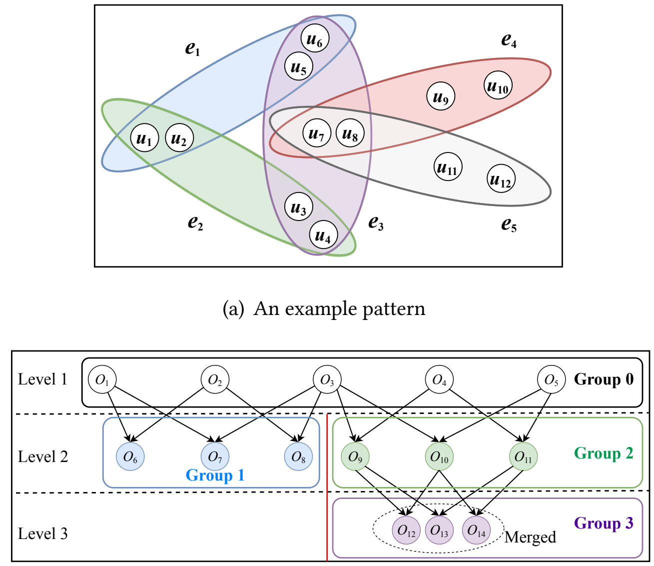
Definition 1 (*Overlap Intersection Graph*). *Given an arbitrary pattern, an OIG is a directed acyclic ordinary graph, where each vertex represents either a hyperedge or an overlap. An edge from src to dst indicates that dst is formed by intersecting src and another vertex that has an outgoing edge directly connected* to *dst*.

Algorithm 1 demonstrates the procedure for constructing an OIG of P using the *Breadth-First Search* (BFS) strategy. A level in OIG, called *Level Vertices* (LV), contains the vertices at the same depth of the BFS. First, all hyperedges of P are added into the *OIG* as vertices of level 1 (lines 1–2), which are stored in *LV* (lines 3–4). Then, we explore each pair of vertices $\{lv_i, lv_j\}$ $(i < j)$ in *LV* (lines 6–8). If lv_i and lv_j are overlapped, we add their overlap (i.e., $overlap_{ij}$) as a vertex and two edges (i.e., $\{lv_i, overlap_{ij}\}$ and $\{lv_j, overlap_{ij}\}$) into *OIG* (lines 8–12). Next, we merge the identical vertices in the next level to avoid redundant intersection computations among them in the remaining levels (line 13) and then store the vertices of the next level to *LV* (line 14). Finally, we continue the preceding steps until *LV* is empty (lines 5–14).

The OIG of the example pattern is shown on the left of Figure 8 and contains three levels. The vertices at level 1 in the OIG represent hyperedges, while the other vertices represent

Algorithm 1: OIG Generation

```
Input: Pattern P
Output: Overlap intersection graph of P
foreach pe_i ∈ P do
    OIG.addVertex(pe_i);
level ← 1;
LV ← GetLevelVertices(OIG, level);
while LV ≠ ∅ do
    foreach lv_i ∈ LV do
        foreach lv_j ∈ LV do
            if i < j and N(lv_i) ∩ N(lv_j) ≠ ∅ then
                overlap_ij ← N(lv_i) ∩ N(lv_j);
                OIG.addVertex(overlap_ij);
                OIG.addEdge(lv_i, overlap_ij);
                OIG.addEdge(lv_j, overlap_ij);
    OIG ← MergeForUnique(OIG, ++level);
    LV ← GetLevelVertices(OIG, level);
return OIG;
```



(a) An example pattern

(b) The OIG of the example pattern

Figure 9. An example pattern and its OIG to illustrate our group-based approach

overlaps. Each vertex in level i (except for level 1) represents an overlap of i hyperedges. Specifically, level 1 represents the hyperedges of the pattern (i.e., pe_1, pe_2, and pe_3), and level 2 consists of the vertices formed by the intersection of two hyperedges. Note that the overlap of pe_1 and pe_2 is the same as the overlap of pe_1 and pe_3, and thus we merge these two overlaps (i.e., the dashed circle). Level 3 contains the final overlap, which is the intersection of two overlaps in level 2. The right in Figure 8 depicts the simplified representation of OIG, where o_1 and o_6 represent the hyperedge pe_1 and the overlap of pe_2 and pe_3, respectively.

To support labeled HPM, we add the label attribute for each vertex in OIG to ensure correctness. For efficiency of labeled HPM, we maintain a sorted sequence of the elements in hyperedges (or overlaps) based on both label ID and vertex ID (label ID first, then vertex ID), rather than only vertex ID in unlabeled HPM, for efficient set operations. Note that our techniques can also be easily extended to hyperedge-labeled hypergraphs. Specifically, we can prune the unrelated hyperedges based on the hyperedge labels when extending a hyperedge, which reduces the search space of HPM and the computation load.

4.3.2 OIG Analysis and Optimization. To match a vertex in OIG, its predecessors must be matched due to data dependencies (e.g., o_4 is the overlap between o_1 and o_2, which have precedence over o_4), and thus there needs a total order of all vertices in OIG. To achieve this goal, we propose the concept of ***overlap order*** to guide the execution of vertices in OIG. We build a matching order for the pattern's hyperedges based on their connection relationships and data hypergraph features [65]. Given an OIG and a matching order of a pattern, an overlap order is a topological order of the OIG based on the matching order. Consider the simplified OIG in Figure 8 with the matching order $\{o_1, o_2, o_3\}$. When extending o_1, we can only match o_1. When extending o_2, we can match o_4 since its predecessors have been matched. Similarly, when extending o_3, we can first match o_5 and o_6 and then match o_7 once its predecessors (o_{45} and o_6) are matched. Therefore, the overlap order of the example OIG is $\{o_1, o_2, o_4, o_3, o_5, o_{45}, o_6, o_7\}$.

More importantly, the OIG does not model empty overlaps, but these empty overlaps have to be computed in partial embeddings to ensure the correctness of HPM according to Equation (1). For example, two hyperedges that do not overlap in the pattern may overlap in the hypergraph after being mapped, which will incur the incorrect computation result of region sizes (as shown in Figure 4(c)), thereby failing to validate partial embeddings. To overcome this issue, we analyze the disconnection relationships between vertices (i.e., empty overlaps) in the OIG.

However, it is unnecessary to determine an OIG vertex's disconnection relationships to all other vertices. We propose a group-based pruning approach, which uses the disconnection relationships of the current level's vertices to prune those of the remaining levels' vertices. Specifically, we first determine all disconnection relationships among hyperedges in level 1. For example, in Figure 9(b), o_5 is connected to both o_3 and o_4, and does not connect to o_1 and o_2. Then, we divide vertices in the same level into groups, where each group's predecessors are connected to each other. In this way, we only need to determine the disconnection relationships within

Table 1. The overlap-centric execution plan, where c_i, $D(c_i)$, and $A(c_i)$ represent a candidate of o_i, degree of c_i, and $adj(c_i)$ in the hypergraph, respectively, for the OIG in Figure 8

Vertex	Overlap-centric Execution Plan
o_1	$c_1 \in E$ && $D(c_1) == D(o_1)$
o_2	$c_2 \in A(c_1)$ && $D(c_2) == D(o_2)$
o_4	$c_4 \leftarrow c_1 \cap c_2$ && $D(c_4) == D(o_4)$
o_3	$c_3 \in (A(c_1) \cap A(c_2))$ && $D(c_3) == D(o_3)$
o_5	$c_5 \leftarrow c_1 \cap c_3$ && $c_5 == c_4$
o_{45}	$c_{45} \leftarrow c_5$
o_6	$c_6 \leftarrow c_2 \cap c_3$ && $D(c_6) == D(o_6)$
o_7	$c_7 \leftarrow c_{45} \cap c_6$ && $D(c_7) == D(o_7)$

each group since the disconnection relationships between groups have already been determined by their predecessors. Consider the two groups (Group 1 and Group 2) in level 2. The disconnection relationships between $\{o_6, o_7, o_8\}$ and $\{o_9, o_{10}, o_{11}\}$ are already determined by the previous level, e.g., the overlap of o_8 and o_9 must be empty since the overlap of o_2 and o_4 is empty.

4.3.3 Execution Plan Generator.

Based on the overlap order and group-based pruning, the generator traverses the OIG to generate the overlap-centric execution plan.

Definition 2 (*Overlap-centric Execution Plan*). *Given an OIG and an overlap order for the pattern, the overlap-centric execution plan is a sequence of set operations and comparison operations, organized in the overlap order, to compute candidates of each vertex in the OIG.*

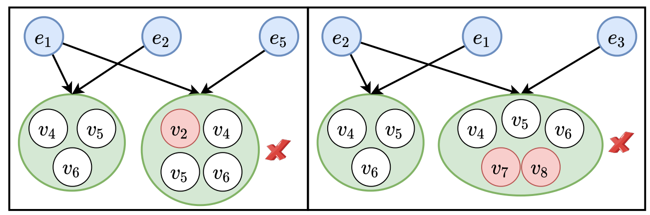
The execution plans for hyperedge vertices and overlap vertices in the OIG guide candidate generation and candidate validation in HPM, respectively. Specifically, to match pe_i, the connection relationships among hyperedges and pe_i's degree determine the execution plan of pe_i, as shown in Figure 1(c). After matching a hyperedge, we compute overlaps in the overlap order. Each overlap is computed by intersecting its predecessor vertices. Table 1 shows the overlap-centric execution plan for the OIG in Figure 8. The execution plan of o_3 specifies that i) o_3's candidate c_3 is connected to both c_1 and c_2, ii) the degree of c_3 (i.e., $D(c_3)$) is the same as the degree of o_3. The execution plan of o_5 specifies that i) o_5's candidate c_5 is the overlap of c_1 and c_3, ii) c_5 is the same as o_4.

4.4 Overlap-centric Parallel Execution Engine

Taking the overlap-centric execution plan and hypergraph data as input, the engine finds all embeddings in the hypergraph. Figure 10 shows the pseudocode of mining the pattern in Figure 1(a). The pseudocode mainly contains two parts: candidate generation for hyperedges (lines 1–4, lines 7–8) and candidate validation for overlaps (lines 5–6, lines 9–14).

```
1. for c1 ∈ E  // candidate generation for o1
2.   if D(c1)! = 6 break;  // candidate generation for o1
3.   for c2 ∈ A(c1)  // candidate generation for o2
4.     if D(c2)! = 6 break;  // candidate generation for o2
5.     c4 = c1 ∩ c2;  // candidate validation for o4
6.     if D(c4)! = 3 break;  // candidate validation for o4
7.     for c3 ∈ (A(c1) ∩ A(c2))  // candidate generation for o3
8.       if D(c3)! = 8 break;  // candidate generation for o3
9.       c45 = c5 = c1 ∩ c3;  // candidate validation for o5
10.      if c5! = c4 break;  // candidate validation for o5
11.      c6 = c2 ∩ c3;  // candidate validation for o6
12.      if D(c6)! = 5 break;  // candidate validation for o6
13.      c7 = c45 ∩ c6;  // candidate validation for o7
14.      if D(c7) == 3  // candidate validation for o7
15.        embeddings ∪= {c1, c2, c3}
```

Figure 10. Pseudocode of mining the pattern in Figure 1(a)



(a) EOIG for $\{e_1, e_2, e_5\}$ (b) EOIG for $\{e_2, e_1, e_3\}$

Figure 11. Examples for overlap-pruned candidate validation when mining the pattern in Figure 1(a)

Overlap-pruned Candidate Validation. To efficiently validate a partial embedding, we incrementally maintain an OIG for a partial embedding (called EOIG) at runtime. Specifically, based on the overlap order, we extend the previous partial EOIG, rather than recompute a new one completely, through computing the newly matched overlaps of EOIG using the execution plan. Once any overlap in EOIG does not meet the conditions for its corresponding overlap in patterns (e.g., degree of overlap, disconnection relationships, and equal relationships, as shown in Table 1), we prune it to avoid subsequent redundant set computations. Consider the process of matching o_5 in Figure 8 when extending $\{e_1, e_2\}$ to $\{e_1, e_2, e_5\}$, as shown in Figure 11(a). The degree of c_5 (i.e., $|e_1 \cap e_5| = 4$) differs from that in the pattern's OIG (i.e., $|pe_1 \cap pe_3| = 3$), and hence it is not a valid embedding of P. Note that if c_5 is not successfully matched, subsequent overlaps (i.e., c_{45}, c_6, and c_7) are not required to be computed. However, HGMatch [65] performs candidate validation without pruning, where it computes and hashes profiles of all vertices in the matched hyperedge and then compares them with those of the pattern.

Parallelism. We adopt OpenMP with dynamic scheduling to implement the thread-level parallelism. Specifically, we assign candidates of the first hyperedge in a pattern to

Table 2. The DAL for the example hypergraph in Figure 1(b)

Hyperedge	Degree	AL	DAL
e_1	6	$\{e_2, e_3, e_4, e_5\}$	$\{\{e_2, e_4\}, \{e_3, e_5\}\}$
e_2	6	$\{e_1, e_3, e_5\}$	$\{\{e_1\}, \{e_3, e_5\}\}$
e_4	6	$\{e_1, e_3\}$	$\{\{e_1\}, \{e_3\}\}$
e_3	8	$\{e_1, e_2, e_4, e_5\}$	$\{\{e_1, e_2, e_4\}, \{e_5\}\}$
e_5	8	$\{e_1, e_2, e_3\}$	$\{\{e_1, e_2\}, \{e_3\}\}$

multiple threads for execution. Each thread computes partial results (e.g. the number of subhypergraphs), and then the main thread is used to synchronize and accumulate the results from all other threads. Each thread explores the search tree (Figure 1(d)) using a *Depth-First Search* (DFS) strategy to minimize the memory consumption and data copying overhead. Moreover, we use SIMD instructions (i.e., AVX-512) to support the data-level parallelism for efficient set computations [26].

4.5 Degree-aware Data Store

The core operation of generating candidates is to determine whether the candidate of pe_i is connected and disconnected to the previously matched hyperedges with the pruning of pe_i's degree. For example, lines 7–8 in Figure 10 show the pseudocode of generating candidates for pe_3. To efficiently support this, we propose a *Degree-aware Adjacency List* (DAL) to directly describe connection relationships of hyperedges in a data hypergraph, rather than the indirect approach in HGMatch [65], which incurs massive redundant computations (see Section 2.3). DAL groups hyperedges according to their degrees to efficiently match the first vertex in the overlap order. For each hyperedge e_i, DAL maintains an ID list of hyperedges connected to it, and then groups this list based on their degrees for efficient pruning.

Table 2 shows that the DAL of e_1 is partitioned into two groups, i.e., $\{e_2, e_4\}$ for degree 6 and $\{e_3, e_5\}$ for degree 8. To generate candidates of pe_3 (degree is 8) based on the partial embedding $\{e_1, e_2\}$ in Figure 1, we first get adjacency lists of e_1 and e_2 with degree 8 since both pe_1 and pe_2 are connected to pe_3. Note that we only need to scan the group $\{e_3, e_5\}$ instead of the entire *Adjacency List* (AL) $\{e_2, e_3, e_4, e_5\}$ for e_1. Then, we intersect degree-pruned adjacency lists of e_1 and e_2 to obtain candidates for pe_3, i.e., $\{e_3, e_5\}$. Note that we only need to retrieve the neighbors of two hyperedges with degree pruning, rather than the incident hyperedges of five vertices, as shown in Figure 2(a).

Like *Compressed Sparse Row* (CSR) [54], we maintain a degree index for the sorted adjacency list (degree first, then hyperedge ID) of each hyperedge. The index is used to locate the start position of the hyperedges that share the same degree. Note that the DAL construction is accomplished by offline preprocessing, and its overhead can be amortized by numerous executions of different HPM applications.

Table 3. A list of real-world hypergraph datasets with their number of vertices, hyperedges, and *Average Degree of hyperedges* (AD) for evaluation

Hypergraph Datasets	|V|	|E|	AD
contact-high-school (CH)	327	7,818	2.33
contact-primary-school (CP)	242	12,704	2.42
senate-bills (SB)	294	29,157	9.90
house-bills (HB)	1,494	60,987	22.15
walmart-trips (WT)	88,860	69,906	6.86
trivago-clicks (TC)	172,738	233,202	3.18
coauth-DBLP (CD)	1,924,991	3,700,067	3.14
AMiner (AM)	13,262,573	22,552,647	3.82

5 Evaluation

5.1 Methodology

Environments. We run all the experiments on a machine equipped with two 32-core Intel Xeon Platinum 8358 CPUs (64 cores and 128 threads in total) and 1024 GB DDR4 RAM. We implement OHMiner using C++ and compile it using the g++ compiler (version 9.4) with O3 optimization.

Hypergraph Datasets and Patterns. We evaluate eight real-world hypergraphs [6, 36], as shown in Table 3. We remove redundant hyperedges and redundant vertices in each hyperedge when preprocessing the hypergraph. The hypergraphs in Table 3 are mainly those used by HGMatch. Additionally, we have conducted experiments with even larger hypergraphs (i.e., CD and AM). CD is an author collaboration network, where a hyperedge (paper) contains many vertices (authors). AM is an academic bibliographic network, representing the relationships that an author (hyperedge) publishes multiple publications (vertices). These datasets vary in |E|, |V|, degree sizes, degree distributions, and |E|/|V|-ratios. Hypergraphs with higher |E|/|V| ratios tend to have more overlaps. Our tests show that some of the datasets (e.g., WT and TC) exhibit a clear power-law distribution, while others do not.

Like HGMatch [65], we randomly sample subhypergraphs from hypergraphs as patterns. We continuously expand a hyperedge sampled from the adjacent hyperedges of previous hyperedges based on the pattern setting. P_i represents a pattern setting that contains i hyperedges, and pi represents a pattern in P_i. The number of vertices is limited to the range $[V_{min}, V_{max}]$, as shown in Table 4. For each pattern setting, we randomly generate 5 patterns by default and report the average result. For each pattern, we measure the average execution time three times.

Baselines. We compare OHMiner with the best-performing HPM system HGMatch [65] to date. Note that HGMatch significantly outperforms the other cutting-edge solutions [8, 9,

Table 4. A list of pattern settings with their pattern numbers, hyperedge numbers, and vertex numbers for evaluation

Pattern Setting	Pattern Number	$\lvert\mathbf{E}\rvert$	$\lvert\mathbf{V}\rvert_{min}$	$\lvert\mathbf{V}\rvert_{max}$
P_2	5	2	5	15
P_3	5	3	10	20
P_4	5	4	10	30
P_5	5	5	15	35
P_6	5	6	15	40

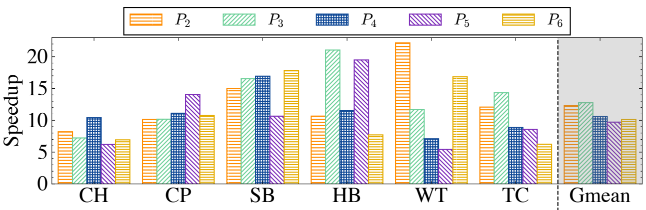


Figure 12. Speedups of OHMiner compared to HGMatch for unlabeled HPM normalized to that of HGMatch

Table 5. Execution times (in seconds) of HGMatch and OHMiner for different patterns and datasets

Patterns	Datasets	HGMatch	OHMiner	Speedup
p_3	SB	1.66	0.13	12.77
p_3	HB	1.35	0.07	19.29
p_3	WT	1.05	0.11	9.55
p_4	SB	74.46	3.31	22.50
p_4	HB	36.32	4.18	8.69
p_4	WT	93.35	9.75	9.57
p_5	SB	1910.77	166.63	11.47
p_5	HB	2980.65	264.56	11.27
p_5	WT	870.04	120.52	7.22

14, 15, 24, 25, 55, 56] by four orders of magnitude on average, because HGMatch uses the match-by-hyperedge approach rather than the match-by-vertex approach [8, 9, 14, 15, 24, 25, 55, 56]. Thus, we only compare with HGMatch in our experiments. Note that, for a fair comparison, we modify HGMatch to employ the parallelism strategy described in Section 4.4, which outperforms HGMatch's fine-grained parallelism strategy [65] for all patterns by 1.2× at least.

5.2 Performance

Overall Speedup. Figure 12 compares the performance of OHMiner and HGMatch for unlabeled HPM. The results show that OHMiner outperforms HGMatch by 8.2×–22.2×, 7.2×–21.0×, 7.1×–17.0×, 5.4×–19.5×, and 6.2×–17.8×, respectively, for different pattern settings. The perfomance improvement stems from two factors. First, the overlap-centric method leverages set computations effectively to eliminate the redundant computations in HGMatch. Second, we adopt

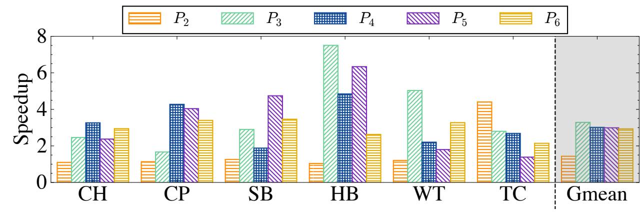


Figure 13. Speedups of OHM-V compared to HGMatch for unlabeled HPM normalized to that of HGMatch

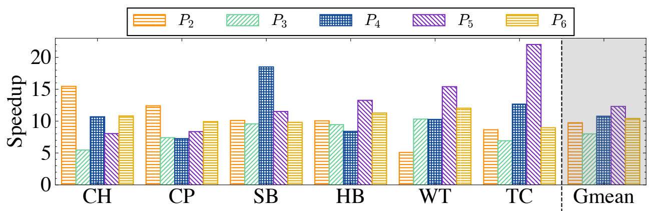


Figure 14. Speedups of OHMiner compared to HGMatch for labeled HPM normalized to that of HGMatch

the DAL and overlap-pruned approach to efficiently generate and validate candidates, respectively, in HPM. Table 5 shows the execution time of HGMatch and OHMiner. For example, HGMatch takes more than 30 minutes to mine a pattern of P_5 on SB, whereas OHMiner only needs less than 3 minutes.

Note that OHMiner's performance improvement does not mainly come from SIMD instructions. Remarkably, even without SIMD instructions, OHMiner's performance exceeds HGMatch by 3.8×–19.6×. With SIMD instructions, OHMiner's performance exceeds HGMatch by 5.4×–22.2×. The reason for the limited improvement from SIMD instructions is primarily due to the short length of sets involved in set computations.

Speedup for Validation. To show the performance improvement brought by candidate validation, we implement OHM-V, which uses HGMatch's candidate generation and OHMiner's candidate validation. We compare OHM-V with HGMatch on different datasets and pattern settings. Figure 13 shows that OHM-V outperforms HGMatch by 1.05×–4.4×, 1.7×–7.5×, 1.9×–4.8×, 1.4×–6.3×, and 2.1×–3.5×, respectively, for different pattern settings. This indicates that OHMiner still achieves a significant performance improvement even when it generates candidates without DAL.

Speedup for Labeled HPM. We also compare OHMiner with HGMatch for labeled HPM. We use a single thread in this set of experiments since labeled HPM with a single thread typically runs below one second, making parallelization less critical. This is because the use of labels can significantly prune the search space. Figure 14 shows that OHMiner outperforms HGMatch by 5.1×–15.5×, 5.5×–10.3×, 7.3×–18.5×, 8.0×–22.0×, and 8.9×–12.0×, respectively, for

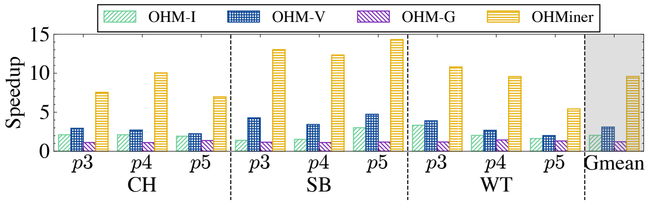


Figure 15. Speedups with different optimization techniques of OHMiner normalized to that of HGMatch

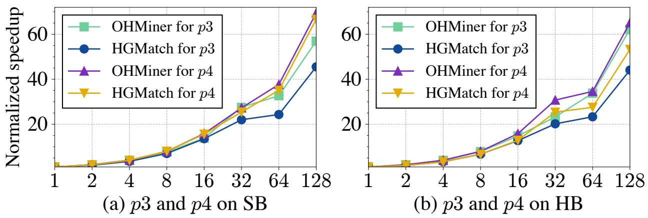


Figure 16. The normalized speedups with different numbers of threads for different patterns

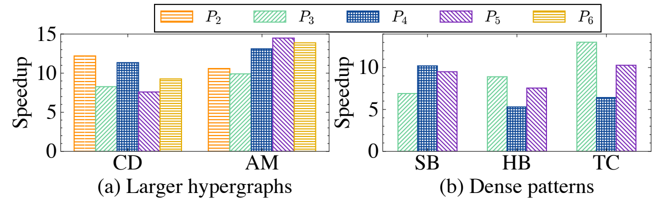


Figure 17. Speedups of OHMiner compared to HGMatch on larger hypergraphs and dense patterns

different pattern settings. The results show that OHMiner also outperforms HGMatch on labeled HPM.

5.3 Ablation Studies about Optimization Techniques

To analyze the performance impact of optimizations, we implement OHM-I and OHM-G, which use HGMatch's candidate generation and OHMiner's IEP optimization and overlap-pruned approach, and OHMiner's candidate generation and HGMatch's candidate validation, respectively. We evaluate their performance on different datasets and patterns. Figure 15 shows that our IEP optimized method (OHM-I) has exceeded HGMatch by 1.40×–3.01×. With our redundancy-free overlap pattern analysis, OHM-V gains more performance improvement and outperforms HGMatch by 2.01×–4.74×. Moreover, OHM-G only outperforms HGMatch by 1.11×–1.45× since the major bottleneck of HGMatch is candidate validation. Using our candidate generation based on OHM-V, OHMiner outperforms OHM-V by 2.56×–3.70×.

5.4 Scalability

Multiple Threads. Figure 16 compares the performance of OHMiner and HGMatch with different numbers of threads. The performance is normalized to their respective single-threaded performances. As the number of threads increases, OHMiner exhibits better scalability than HGMatch. For example, when performing $p3$ on HB with 128 threads, OHMiner improves performance by 62.2× compared to that of its single-threaded counterpart, while HGMatch improves performance by only 44.1× compared to its single-threaded performance. In other words, with a single thread, OHMiner surpasses HGMatch by 15.2× due to our overlap-centric execution model adopting various optimizations. With 128 threads, OHMiner further outperforms HGMatch by 21.5× due to OHMiner's better multi-threading scalability than HGMatch. The main reason is that OHMiner supports HPM using efficient set operations, rather than traversing hyperedges as well as these hyperedges' vertices to fetch incident hyperedges of numerous vertices as in HGMatch [65], which leads to significant memory access overhead.

Larger Hypergraphs. We evaluate the HPM performance of OHMiner on larger hypergraph datasets, i.e., CD and AM, as shown in Table 3. CD contains 3.7 million hyperedges, and AM contains 22.5 million hyperedges. We compare OHMiner against HGMatch by evaluating them with various pattern settings. Figure 17(a) shows that OHMiner outperforms HGMatch on CD and AM by 7.6×–12.2× and 9.9×–14.5×, respectively. This demonstrates that OHMiner works better with hypergraph datasets than HGMatch, enabling it to effectively process large data in complicated real-world scenarios. Note that hypergraphs with tens of millions of hyperedges are the largest real-world hypergraphs that are currently published. We synthesize a hypergraph with 100 million hyperedges using the similar method from [53]. Experimental results show that OHMiner outperforms HGMatch by 7.9×–20.1×.

5.5 Sensitivity Study

To analyze the efficiency of OHMiner with more overlaps in patterns, in which OHMiner needs more set computations, we generate and evaluate various dense patterns, where each hyperedge is connected to all other hyperedges. We compare the performance of OHMiner with that of HGMatch on different hypergraph datasets. As shown in Figure 17(b), OHMiner outperforms HGMatch on SB, HB, and TC by 6.9×–10.2×, 5.3×–8.9×, and 6.4×–13.0×, respectively. Note that Figure 17(b) aims to show the performance of OHMiner and HGMatch for dense patterns. Only the datasets, SB, HB, and TC, are used because they are the datasets that have densely connected hyperedges and have more dense patterns. This result demonstrates that our approach outperforms HGMatch significantly, even in scenarios where OHMiner needs lots of set computations for computing overlaps.

5.6 Overhead

Time of Compiling Pattern. The time of compiling pattern (called OIG-T) is significantly small. As shown in Table 6, although we evaluate a pattern with 6 hyperedges (patterns

Table 6. Overheads of OHMiner (OIG-T, DAL-T, DAL-M, HGMatch-M, and HPM-T represent the time of compiling pattern, the time of DAL, the memory usage of DAL, the memory usage of HGMatch, and the time of HPM)

Datasets	OIG-T	DAL-T	DAL-M	HGMatch-M	DAL-T/HPM-T
CH	0.04 ms	0.02 s	4.5 MB	207 KB	0.1%
CP	0.05 ms	0.07 s	16.8 MB	342 KB	0.1%
SB	0.07 ms	1.19 s	48.4 MB	1.89 MB	0.9%
HB	0.08 ms	5.58 s	2.50 GB	10.7 MB	3.4%
WT	1.73 ms	0.89 s	239 MB	5.07 MB	1.4%
TC	1.41 ms	0.41 s	44 MB	8.61 MB	1.3%
CD	1.78 ms	3.77 s	1.01 GB	92.7 MB	1.9%
AM	0.81 ms	5.83 s	1.25 GB	547 MB	2.0%

with more hyperedges have larger overheads, and patterns with the same number of hyperedges have similar overheads, thus we evaluate a pattern with 6 hyperedges, which is the biggest number of hyperedges in the sampled patterns), OIG-T only ranges from 0.04 milliseconds to 1.85 milliseconds for different hypergraphs. Compared with the time for HPM, which often takes several seconds or minutes, this overhead can be negligible.

Construction Time and Memory Footprint of DAL. Table 6 shows that the DAL construction times (DAL-T) on different hypergraphs range from 0.03 seconds to 5.83 seconds, which accounts for only 0.1%–3.4% of the total time of HPM (DAL-T/HPM-T). Note that each hypergraph's DAL is constructed only once and can be reused by different HPM applications. Even when OHMiner does not use DAL to generate candidates, it can still outperform HGMatch by up to 7.5 times, as shown in Figure 13. In addition, the storage space of DAL (DAL-M) ranges from 4.5 MB to 2.50 GB, which is rather small given that the total memory size of a typical server is large (the memory size of the server in our experiments is 1024 GB).

6 Related Work

Graph Pattern Mining. Prior GPM systems [1, 17, 20, 21, 57, 60, 61] employ a *pattern-oblivious* approach, which constantly expands partial embeddings by adding a vertex and then examines if the final embeddings are isomorphic to the pattern. However, it suffers from extensive isomorphism tests and redundant explorations. To address these issues, recent GPM systems [8, 9, 16, 18, 19, 23, 25, 31–33, 43, 44, 51, 52, 56] use a *pattern-aware* approach, which adopts the matching order to eliminate isomorphism testing and the symmetric order to prune exploration space. Note that Tesseract [10] and PSMiner [48] focus on mining subgraphs in dynamic graphs, where the vertices and edges are constantly updated [7, 35, 47]. Moreover, ASAP [30] and Arya [67] leverage sampling-based approximation techniques to estimate the occurrences of patterns.

However, these systems are designed and optimized for GPM and cannot directly support HPM. Hypergraphs [13, 22, 28, 29, 34, 37, 38, 40, 41, 45, 49, 65] are common in many modern applications and offer a more natural representation for capturing relationships involving more than two entities. Due to the lack of efficient hypergraph-oriented graph processing systems, existing systems often convert hypergraphs into ordinary graphs for processing, which loses the advantage of using hypergraphs to concisely represent multi-entity relationships [53]. OHMiner addresses this gap.

Hypergraph Pattern Mining. Existing studies can be classified into two categories: match-by-vertex [14, 15, 24, 55, 66] and match-by-hyperedge [65]. The former continuously maps a pattern vertex to a data vertex in hypergraphs and then validates hyperedges. Although several optimization techniques are proposed (e.g., structural indexing [66] and pruning with hyperedge features [24, 55]), this approach still suffers from large search space and expensive enumeration costs. To reduce these issues, HGMatch [65] presents a match-by-hyperedge approach, which extends a subhypergraph by adding a hyperedge and validates the extended subhypergraph in each extension. However, HGMatch suffers from massive redundant computations because it adopts a vertex-granularity approach to repeatedly fetch and process the same incident hyperedges for different vertices. OHMiner designs an overlap-centric execution model with many optimizations, which efficiently handles vertices collectively that share the same incident hyperedges, and therefore accelerates the processing of HPM. Note that several systems [53, 62, 63] are developed for hypergraph processing, which alternatively computes the states of hyperedges and vertices for many rounds, and cannot support HPM.

7 Conclusion

We develop a novel overlap-centric system OHMiner for high-performance HPM. OHMiner introduces an overlap-centric execution model to efficiently validate subhypergraph isomorphism. Based on this model, the compiler of OHMiner analyzes the overlapping semantics of patterns by constructing an overlap intersection graph and then generates an overlap-centric execution plan to guide the procedure of HPM, aiming to eliminate redundant set computations. The execution engine of OHMiner further proposes an incremental overlap-pruned technique to rapidly validate candidates for HPM. The data store of OHMiner designs degree-aware adjacency list to efficiently generate candidates. The experimental results show that OHMiner surpasses the cutting-edge HPM system HGMatch by up to 22.2 times.

Acknowledgments

The authors would like to thank our shepherd Keval Vora and all anonymous reviewers for their insightful comments. Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper. This paper is supported by National Key Research and Development Program of China (No. 2024YFB4504200), Key Research and Development Program of Hubei Province (No. 2023BAB078). This work was supported by Ant Group through CCF-Ant Research Fund (No. CCF-AFSG RF20240204).

References

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*. 716–727.

[2] Md Tanvir Alam, Chowdhury Farhan Ahmed, Md Samiullah, and Carson Kai-Sang Leung. 2023. Discovering Interesting Patterns from Hypergraphs. *ACM Transactions on Knowledge Discovery from Data* 18, 1 (2023), 32:1–32:34.

[3] László Babai and Paolo Codenotti. 2008. Isomorhism of Hypergraphs of Low Rank in Moderately Exponential Time. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science*. 667–676.

[4] Federico Battiston, Giulia Cencetti, Iacopo Iacopini, Vito Latora, Maxime Lucas, Alice Patania, Jean-Gabriel Young, and Giovanni Petri. 2020. Networks beyond pairwise interactions: Structure and dynamics. *Physics Reports* 874 (2020), 1–92.

[5] Robert A. Beeler. 2015. *How to Count: An Introduction to Combinatorics and Its Applications*. Springer, Switzerland.

[6] Austin R. Benson. 2022. Hypergraph datasets. https://www.cs.cornell.edu/~arb/data/.

[7] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2023. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1860–1876.

[8] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[10] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: distributed, general graph pattern mining on evolving graphs. In *Proceedings of the 16th European Conference on Computer Systems*. 458–473.

[11] Kellog S. Booth and Charles J. Colbourn. 1979. *Problems Polynomially Equivalent to Graph Isomorphism*. Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada.

[12] Alain Bretto. 2013. *Hypergraph Theory*. Springer Cham, Switzerland.

[13] Alain Bretto, Hocine Cherifi, and Driss Aboutajdine. 2002. Hypergraph imaging: an overview. *Pattern Recognition* 35, 3 (2002), 651–658.

[14] Horst Bunke, Peter Dickinson, and Miro Kraetzl. 2005. Theoretical and Algorithmic Framework for Hypergraph Matching. In *Proceedings of the 13th International Conference on Image Analysis and Processing*. 463–470.

[15] Horst Bunke, Peter Dickinson, Miro Kraetzl, Michel Neuhaus, and Marc Stettler. 2008. *Applied Pattern Recognition*. Springer Berlin Heidelberg, Berlin, Heidelberg.

[16] Joanna Che, Kasra Jamshidi, and Keval Vora. 2024. Contigra: Graph Mining with Containment Constraints. In *Proceedings of the 19th European Conference on Computer Systems*. 50–65.

[17] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Proceedings of the 13th European Conference on Computer Systems*. 32:1–32:12.

[18] Jingji Chen and Xuehai Qian. 2022. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 47–61.

[19] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and scalable distributed graph pattern mining engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 413–426.

[20] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1190–1205.

[21] Vinícius Vitor dos Santos Dias, Carlos H. C. Teixeira, Dorgival O. Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. 2019. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*. 1357–1374.

[22] Song Feng, Emily Heath, Brett Jefferson, Cliff Joslyn, Henry Kvinge, Hugh D. Mitchell, Brenda Praggastis, Amie J. Eisfeld, Amy C. Sims, Larissa B. Thackray, Shufang Fan, Kevin B. Walters, Peter J. Halfmann, Danielle Westhoff-Smith, Qing Tan, Vineet D. Menachery, Timothy P. Sheahan, Adam S. Cockrell, Jacob F. Kocher, Kelly G. Stratton, Natalie C. Heller, Lisa M. Bramer, Michael S. Diamond, Ralph S. Baric, Katrina M. Waters, Yoshihiro Kawaoka, Jason E. McDermott, and Emilie Purvine. 2021. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. *BMC Bioinformatics* 22, 1 (2021), 287–207.

[23] Chuangyi Gui, Xiaofei Liao, Long Zheng, and Hai Jin. 2023. Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow. In *Proceedings of the 2023 USENIX Annual Technical Conference*. 71–85.

[24] Tae Wook Ha, Jung Hyuk Seo, and Myoung Ho Kim. 2018. Efficient Searching of Subhypergraph Isomorphism in Hypergraph Databases. In *Proceedings of the 2018 IEEE International Conference on Big Data and Smart Computing*. 739–742.

[25] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[26] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data*. 1587–1602.

[27] Yi Han, Bin Zhou, Jian Pei, and Yan Jia. 2009. Understanding importance of collaborations in co-authorship networks: A supportiveness analysis approach. In *Proceedings of the 2009 SIAM International Conference on Data Mining*. 1112–1123.

[28] Tianming Hu, Hui Xiong, Wenjun Zhou, Sam Yuan Sung, and Hangzai Luo. 2008. Hypergraph Partitioning for Document Clustering: A Unified Clique Perspective. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 871–872.

[29] TaeHyun Hwang, Ze Tian, Rui Kuangy, and Jean-Pierre Kocher. 2008. Learning on Weighted Hypergraphs to Integrate Protein Interactions and Gene Expressions for Cancer Outcome Prediction. In *Proceedings of the 2008 IEEE International Conference on Data Mining*. 293–302.

[30] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. 745–761.

[31] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the 15th European Conference on Computer Systems*. 13:1–13:16.

[32] Kasra Jamshidi and Keval Vora. 2021. A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 1–10.

[33] Kasra Jamshidi, Harry Xu, and Keval Vora. 2023. Accelerating Graph Mining Systems with Subgraph Morphing. In *Proceedings of the 18th European Conference on Computer Systems*. 162–181.

[34] Ping Jian, Keming Chen, and Chenwei Zhang. 2016. A hypergraph-based context-sensitive representation technique for VHR remote-sensing image change detection. *International Journal of Remote Sensing* 37, 8 (2016), 1814–1825.

[35] Hai Jin, Hao Qi, Jin Zhao, Xinyu Jiang, Yu Huang, Chuangyi Gui, Qinggang Wang, Xinyang Shen, Yi Zhang, Ao Hu, Dan Chen, Chaoqiang Liu, Haifeng Liu, Haiheng He, Xiangyu Ye, Runze Wang, Jingrui Yuan, Pengcheng Yao, Yu Zhang, Long Zheng, and Xiaofei Liao. 2022. Software Systems Implementation and Domain-Specific Architectures towards Graph Analytics. *Intelligent Computing* 2022 (2022), 1–32.

[36] Sunwoo Kim, Dongjin Lee, Yul Kim, Jungho Park, Taeho Hwang, and Kijung Shin. 2023. Datasets, tasks, and training methods for large-scale hypergraph learning. *Data Mining and Knowledge Discovery* 37, 6 (2023), 2216–2254.

[37] Steffen Klamt, Utz-Uwe Haus, and Fabian Theis. 2009. Hypergraphs and Cellular Networks. *PLoS Computational Biology* 5, 5 (2009), 1–6.

[38] Dan Klein and Christopher D. Manning. 2001. Parsing and Hypergraphs. In *Proceedings of the 7th International Workshop on Parsing Technologies*. 123–134.

[39] David Knoke and Song Yang. 2019. *Social Network Analysis*. SAGE publications, Thousand Oaks, California, USA.

[40] Lei Li and Tao Li. 2013. News Recommendation via Hypergraph Learning: Encapsulation of User Behavior and News Content. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*. 305–314.

[41] Xi Li, Yao Li, Chunhua Shen, Anthony Dick, and Anton Van Den Hengel. 2013. Contextual hypergraph modeling for salient object detection. In *Proceedings of the 2013 IEEE International Conference on Computer Vision*. 3328–3335.

[42] Anna Lubiw. 1981. Some NP-complete problems similar to graph isomorphism. *SIAM J. Comput.* 10, 1 (1981), 11–21.

[43] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 21–37.

[44] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

[45] Telmo Menezes and Camille Roth. 2019. Semantic Hypergraphs. arXiv:1908.10784

[46] Daniel Neuen. 2022. Hypergraph isomorphism for groups with restricted composition factors. *ACM Transactions on Algorithms* 18, 3 (2022), 27:1–27:50.

[47] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: A Locality-centric High-performance Streaming Graph Engine. In *Proceedings of the 19th European Conference on Computer Systems*. 33–49.

[48] Hao Qi, Yu Zhang, Ligang He, Kang Luo, Jun Huang, Haoyu Lu, Jin Zhao, and Hai Jin. 2023. PSMiner: A Pattern-Aware Accelerator for High-Performance Streaming Graph Pattern Mining. In *Proceedings of the 60th ACM/IEEE Design Automation Conference*. 1–6.

[49] Emad Ramadan, Arijit Tarafdar, and Alex Pothen. 2004. A Hypergraph Model for the Yeast Protein Complex Network. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. 189–196.

[50] Frank Ruskey and Mark Weston. 2005. A Survey of Venn Diagrams. https://www.combinatorics.org/files/Surveys/ds5/ds5v3-2005/VennEJC.html.

[51] Tianhui Shi, Jidong Zhai, Haojie Wang, Qiqian Chen, Mingshu Zhai, Zixu Hao, Haoyu Yang, and Wenguang Chen. 2023. GraphSet: High Performance Graph Mining through Equivalent Set Transformations. In *Proceedings of the 2023 International Conference for High Performance Computing, Networking, Storage and Analysis*. 32:1–32:14.

[52] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *Proceedings of the 2020 International Conference for High Performance Computing, Networking, Storage and Analysis*. 100:1–100:14.

[53] Julian Shun. 2020. Practical Parallel Hypergraph Algorithms. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 232–249.

[54] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–146.

[55] Yuhang Su, Yu Gu, Zhigang Wang, Ying Zhang, Jianbin Qin, and Ge Yu. 2023. Efficient Subhypergraph Matching Based on Hyperedge Features. *IEEE Transactions on Knowledge and Data Engineering* 35, 6 (2023), 5808–5822.

[56] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[57] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[58] Leo Torres, Ann S. Blevins, Danielle Bassett, and Tina Eliassi-Rad. 2021. The why, how, and when of representations for complex systems. *SIAM Rev.* 63, 3 (2021), 435–485.

[59] Loc Hoang Tran and Linh Hoang Tran. 2015. Hypergraph and Protein Function Prediction with Gene Expression Data. *Journal of Automation and Control Engineering* 3, 2 (2015), 164–170.

[60] Vasileios Trigonakis, Jean-Pierre Lozi, Tomás Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 209–224.

[61] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. 763–782.

[62] Qinggang Wang, Long Zheng, Ao Hu, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. 2022. A Data-Centric Accelerator for High-Performance Hypergraph Processing. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture*. 1326–1341.

,

[63] Qinggang Wang, Long Zheng, Jingrui Yuan, Yu Huang, Pengcheng Yao, Chuangyi Gui, Ao Hu, Xiaofei Liao, and Hai Jin. 2022. Hardware-Accelerated Hypergraph Processing with Chain-Driven Scheduling. In *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture*. 184–198.

[64] Dingqi Yang, Bingqing Qu, Jie Yang, and Philippe Cudre-Mauroux. 2019. Revisiting User Mobility and Social Relationships in LBSNs: A Hypergraph Embedding Approach. In *Proceedings of the 2019 International Conference on World Wide Web*. 2147–2157.

[65] Zhengyi Yang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Shunyang Li. 2023. HGMatch: A Match-by-Hyperedge Approach for Subgraph Matching on Hypergraphs. In *Proceedings of the 39th IEEE International Conference on Data Engineering*. 2063–2076.

[66] Xinran Yu and Turgay Korkmaz. 2016. Hypergraph querying using structural indexing and layer-related-closure verification. *Knowledge and Information Systems* 46, 3 (2016), 537–565.

[67] Zeying Zhu, Kan Wu, and Zaoxing Liu. 2023. Arya: arbitrary graph pattern mining with decomposition-based sampling. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*. 1013–1030.