HGMatch: A Match-by-Hyperedge Approach for Subgraph Matching on Hypergraphs

Zhengyi Yang[‡], Wenjie Zhang[‡], Xuemin Lin[§]*, Ying Zhang[♮], Shunyang Li[‡]

[‡]*University of New South Wales,* [§]*Shanghai Jiao Tong University,* [♮]*University of Technology Sydney*
Corresponding Author
[‡]{zyang,zhangw,sli}@cse.unsw.edu.au, [§]xuemin.lin@sjtu.edu.cn, [♮]ying.zhang@uts.edu.au

Abstract—**Hypergraphs are a generalisation of graphs in which a hyperedge can connect any number of vertices. It can describe n-ary relationships and high-order information among entities compared to conventional graphs. In this paper, we study the fundamental problem of subgraph matching on hypergraphs (i.e., subhypergraph matching). Existing methods directly extend subgraph matching algorithms to the case of hypergraphs. However, this approach delays hyperedge verification and underutilises the high-order information in hypergraphs, which leads to large search space and high enumeration costs. Furthermore, with the growing size of hypergraphs, it is becoming hard to compute subhypergraph matching sequentially. Thus, we propose an efficient and parallel subhypergraph matching system, HGMatch, to handle subhypergraph matching in massive hypergraphs. We propose a novel match-by-hyperedge framework to utilise high-order information in hypergraphs and use set operations for efficient candidate generation. Moreover, we develop an optimised parallel execution engine in HGMatch based on the dataflow model, which features a task-based scheduler and fine-grained dynamic work stealing to achieve bounded memory execution and better load balancing. Experimental evaluation on 10 real-world datasets shows that HGMatch outperforms the extended version of the state-of-the-art subgraph matching algorithms (CFL, DAF, CECI, and RapidMatch) by orders of magnitude when using a single thread, and achieves almost linear scalability when the number of threads increases.**

I. INTRODUCTION

Hypergraphs are a generalisation of graphs in which a hyperedge can connect any number of vertices. In contrast, an edge in conventional graphs (i.e., pairwise graphs) connects exactly two vertices. Compared to conventional graphs, hypergraphs can describe n-ary relationships among entities. Thus, hypergraphs are able to capture high-order interactions among multiple entities that are not directly expressible in conventional graphs.

One of the most fundamental problems in hypergraphs is *subhypergraph matching*. Specifically, given a query hypergraph and a data hypergraph, subhypergraph matching aims to find all subhypergraphs of the data hypergraph that are isomorphic to the query hypergraph. For example, given a sample query hypergraph q and a data hypergraph H in Fig. 1. Subhypergraph matching finds two embeddings, which contains hyperedges $\{e_1, e_3, e_5\}$ and $\{e_2, e_4, e_6\}$ in H.

Applications Subhypergraph matching is associated with many applications in the real world. We list four representative examples in different domains. Other applications include object detection in computer vision [14], [43], [55], complex pattern search in collaborative network [34], [86], finding user alignment in social networks [73], and image retrieval [18].
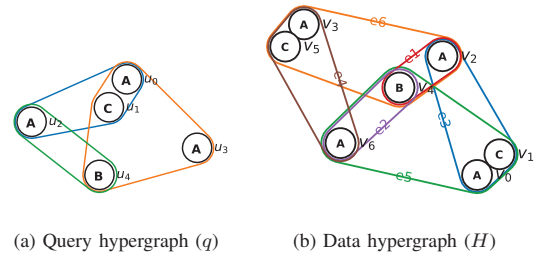

(a) Query hypergraph (q)   (b) Data hypergraph (H)

Fig. 1: Example of Subhypergraph Matching. We draw a "rubber band"-like hypergraph, where convex hulls represent hyperedges and are drawn around the vertices they contain. 'A', 'B' and 'C' represent vertex labels. v_i and e_i represent vertex and hyperedge IDs, respectively.

Mining Biological Networks. Representing biological networks as graphs is a powerful approach to reveal underlying patterns and critical components from biomolecular data. However, conventional graphs do not natively capture the multi-way relationships present among genes and proteins in biological systems [23]. Therefore, hypergraphs are usually used to represent complex high-order relationships in bioinformatics. For example, protein interactions can be represented as hypergraphs where proteins are vertices and complexes are hyperedges [48], [63]. Similarly, gene transcriptomic expression data can also be represented as hypergraphs, with each gene modelled as a hyperedge and each condition as a vertex [23], [40], [74]. By adopting the hypergraph model, biologists can represent protein/gene complexes of their interests as query hypergraphs and find them in large biological networks to understand their interactions and roles.

Querying Hypergraph Databases. In recent years, there emerges a number of hypergraph databases, including AtomSpace [60], HyperGraphDB [42], and TypeDB [76], which rely on subhypergraph matching as a basic building block. A hypergraph database can effectively model the multi-way relations among many real-world entities. Many hypergraph databases are developed and deployed, especially in the area of artificial general intelligence (AGI) [28] (e.g., the OpenCog project [61]). Specifically, AGI systems require a knowledge representation database to support critical reasoning. In a typical AGI application, knowledge is stored in the form of hypergraphs [77], where each vertex and hyperedge represents an *Atom* [36] with a certain type. A pattern matcher, which performs subhypergraph matching, is used to search
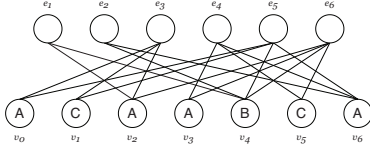
Fig. 2: Converted Bipartite Graph (Data Hypergraph in Fig. 1b).

for specific patterns in the hypergraphs. After specifying some arrangements of atoms (i.e., a query hypergraph), the pattern matcher will find all instances of that hypergraph in the atom space (i.e. a data hypergraph). The matched results can then be sent to a rule engine [6], [81] for further reasoning.

Pattern Learning in NLP. Hypergraphs are also increasingly popular in machine leaning [3], [25], [79], [87] and natural language processing (NLP) [38], [49], [57]. The authors of [57] propose the concept of *semantic hypergraphs* where each word is a vertex, and each valid sentence is a hyperedge. Semantic hypergraphs can be constructed by parsing large corpus using modern machine learning techniques in NLP. During the process of pattern learning, some sentences are first selected from a given training corpus. It can be drawn at random or by any other criterion adapted to the pattern-learning task at hand. The selected sentences are inferred and transformed into a hypergraph query. Subhypergraph matching is then performed in the semantic hypergraph to find matched embeddings. Finally, the embeddings are presented to humans for validation of the corresponding learning tasks. The process repeats with a human-refined query hypergraph if no valid embeddings are found.

Q/A over Hypergraph Knowledge Base. It is observed in [82] that more than 33% of the entities participate in non-binary relations in the knowledge base Freebase [13], and further observed in [22] that 61% of the entities participate in non-binary relations. Question answering (Q/A) allows users to query real-world questions over the knowledge base. By representing the knowledge base in hypergraphs, it allows us to better express and explore the massive non-binary relations in the knowledge base, where the evaluation of queries can be performed using subhypergraph matching [37]. We present a case study of in Section VII-D.

Motivations. Subgraph matching in conventional graphs has been extensively studied in the literature. Existing algorithms of subgraph matching [8], [9], [21], [31], [33], [35], [44], [66], [70], [71] primarily works on better matching orders, pruning rules, index structures, and enumeration methods, to improve the efficiency. However, subhypergraph matching in hypergraphs has attracted little attention despite its emerging applications, as mentioned above.

One straightforward approach of subhypergraph matching is to convert the hypergraph to a *bipartite graph* by treating the hyperedges as the vertices [48]. An example bipartite graph of the data hypergraph in Fig. 1b is given in Fig. 2, where the upper vertices refer to hyperedges in the original hypergraph, the lower vertices refer to vertices in the original hypergraph, and the edges refer to the connectivity of hyperedges. After converting both the query and data hypergraphs into bipar-

tite graphs, conventional subgraph matching algorithms can be applied to find embeddings of the original hypergraph. However, this strawman approach will significantly inflate the size of the graphs. For example, a hypergraph with 2 million vertices and 15 million hyperedges will result in a bipartite graph with 17 million vertices and 1 billion edges [83]. Due to the NP-hardness [26] nature of subgraph matching, it is hard to compute embeddings on such inflated graphs [39].

Another approach is to directly extend existing subgraph matching algorithms to the case of hypergraphs. Among the existing algorithms, [15], [16] extends Ullmann's backtracking algorithm [75]. [30], [68] also follow the same framework with more filtering rules derived from hypergraph features to improve efficiency. As most state-of-the-art subgraph matching algorithms (e.g., [8], [9], [31]) follows Ullmann's backtracking framework, such an extension can be orthogonally applied to them as well. Specifically, they recursively expand partial embedding vertex-by-vertex by mapping a query vertex to a data vertex at each step to enumerate all results following a given matching order and backtrack when necessary. We denote this framework as the *match-by-vertex* approach. Hyperedges are used as a verification condition in the match-by-vertex framework, just like the edges in subgraph matching. However, treating hyperedges simply as verification conditions delays the hyperedge verification and underutilises the high-order information in hypergraphs, which can lead to a huge search space and large enumeration cost.

In addition, with the rapid growth of hypergraph data these days, it is becoming difficult to compute subhypergraph matching on massive hypergraphs using sequential algorithms. However, none of the existing subhypergraph matching algorithms supports parallel execution. For example, in the real-world hypergraph of Amazon Reviews with more than 4 million hyperedges (AR in Table II), none of the existing sequential algorithms in our experiments are able to compute queries of three hyperedges within a one-hour time limit. Traditional backtracking framework in subgraph matching adopts the depth-first search (DFS), which is generally hard to parallelise. Distributed solutions of subgraph matching [2], [5], [50], [51], [62], [72], on the other hand, adopt the breath-first search (BFS) in a cluster of machines for high CPU utilisation. But this can often lead to high memory consumption, network communications, and economic cost [56].

Motivated by the above reasons, we study the problem of subhypergraph matching to develop an efficient and parallel solution on a single machine in this paper.

Challenges. We summarise key challenges as follows.

1) *How to effectively utilise high-order information in hypergraphs?* Hypergraphs contain n-ary relationships in hyperedges that are typically not presented in conventional graphs. Hence, it is crucial to fully utilise the high-order information in hyperedges during matching to reduce search space and speed up enumeration.

2) *How to efficiently enumerate all embeddings in parallel?* To improve the performance of subhypergraph matching on a single machine, it is important to fully utilise the ever-developing hardware (i.e., multi-core) while managing

2064

memory consumption well. Furthermore, the challenge arises from the power law nature of the real-world graphs [19], [20] to handle workload disparity among different workers while parallelising.

Our Solution and Contributions. To address these challenges, we develop HGMatch, an efficient and parallel subHyperGraph Matching engine on a single machine. Instead of matching the query hypergraph vertex-by-vertex as in the match-by-vertex framework used by existing subgraph matching and subhypergraph matching algorithms, we propose a *match-by-hyperedge* framework to match the query by hyperedges to fully utilise the n-ary relationships. Specifically, we made the following contributions.

1) *A match-by-hyperedge framework.* We propose to match the query hypergraph by hyperedges instead of vertices. HGMatch expands a partial embedding by one new hyperedge at a time. In this way, HGMatch is able to fully utilise the high-order information in hypergraphs to reduce search space and avoid redundant computation of enumerating matchings of vertices. We store the data hypergraph in multiple tables with different *hyperedge signatures* (i.e., a multiset[1] of vertex labels contained in a hyperedge). A lightweight inverted hyperedge index is then built for each table to speed up the retrieve process of incident hyperedges of a given vertex. By doing so, HGMatch is able to generate candidate hyperedges directly using set operations (i.e., difference, union and intersections), which can be implemented very efficiently on modern hardware [32], [41], [54], [78]. Apart from that, we use set comparison to remove false positives during enumeration, which completely avoids expensive recursive calls in traditional backtracking-based enumeration methods.

2) *A highly optimised parallel execution engine.* Thanks to the above-mentioned design, HGMatch does not incur any recursive calls or build any auxiliary structures during runtime, which makes it easy to be parallelised. We adopt the dataflow model [4], [46] for parallel execution in HGMatch, which has been employed in many recent subgraph matching solutions [5], [45], [52], [84]. To bound memory consumption while keeping a high degree of parallelism, we design a task-based scheduler in HGMatch. With the scheduler, we prove that HGMatch achieves a tight memory bound of $O(\overline{a_q} \times |E(q)|^2 \times |E(H)|)$ for subhypergraph matching, where $\overline{a_q}$ is the average arity (i.e., hyperedge size) of query, $|E(q)|$ and $|E(H)|$ are the number of query and data hyperedges, respectively. Furthermore, the dynamic work-stealing mechanism [11], [12], [29] is employed for fine-grained load balancing.

3) *In-depth experiments using real-world datasets.* We conducted extensive experiments on 10 real-world datasets. Results show the efficiency and scalability of HGMatch. Comparing with the extended version of three state-of-the-art subgraph matching algorithms CFL [9], DAF [31], CECI [8], and RapidMatch [71], HGMatch achieves an average

speedup of more than 5 orders of magnitude. When using multi threads, HGMatch achieves almost linear scalability when increasing the number of threads with near-perfect load balancing. Besides, HGMatch is the only algorithm that is able to complete all queries within the time limit.

Paper Organization. The rest of this paper is organized as follows. Section II discusses related work. Section III introduces problem definition and background. In Section IV, we present the workflow and hypergraph storage of HGMatch. We introduce our match-by-hypergraph framework in Section V and the design of our parallel execution engine in Section VI, respectively. Experimental evaluation and case study are presented in Section VII, followed by conclusion in Section VIII.

II. RELATED WORK

Sequential Subgraph Matching. In sequential subgraph matching, the study was initiated by Ullmann's backtracking algorithm [75], which recursively matches query vertices to data vertices following a given matching order and backtracks when necessary. The state-of-the-art algorithms [8], [9], [21], [31], [33], [35], [66] mostly follow the backtracking framework with different matching orders, pruning rules, auxiliary data structures, and enumeration methods. In particular, Turbo$_{ISO}$ [33] proposes to compress the query graph by merging equivalent query vertices with a *CR* auxiliary data structure to speed up the computation. CFL [9] proposes the core-forest-leaf decomposition to reduce redundant Cartesian products and introduces a more compact auxiliary structure *CPI* to solve the exponential size of *CR*. DAF [31] designs a new auxiliary structure *CS* based on the directed acyclic graph (DAG) of the query graph and uses pruning by failing sets to increase the pruning power. Note that HGMatch does not build any auxiliary structure during runtime, leading to low memory cost and easy parallelism. A more recent solution, RapidMatch [71] integrates the backtracking-base exploration method with worst-case optimal join. Surveys and experiments of representative algorithms have been conducted in [53], [70].

Parallel Subgraph Matching. PGX.ISO [64] runs subgraph matching in breath-first search (BFS) to carry out parallel execution. As a result, it requires materialising all intermediate results at each step, which leads to exponential memory consumption. Moreover, load balancing is achieved by copying intermediate results into single global storage, which leads to further cost of copying and synchronization. PSM [69] proposes a generic framework for parallelising recursive backtracking-based subgraph matching algorithms. It represents the problem as a tree search in the state space and different matching algorithms as different orders in the search. CECI [8] uses embedding clusters as the auxiliary structure to better accommodate parallel processing with improved load-balancing. EmptyHeaded [1] and Graphflow [59] employ the multi-way join model to compute subgraph matching using join operations in parallel. Due to the increasing size of graphs, distributed algorithms have also been widely studied [2], [5], [50]–[52], [62], [65], [67], [72], [80], [84] in recent years, which typically focus on unlabeled graphs.

---

[1] A multiset (i.e., bag) is a set that allows for multiple instances for each of its elements.

Subhypergraph Matching Compared with subgraph matching, the research on subhypergraph matching algorithms is rather limited. [15], [16] extend the framework of Ullmann's backtracking algorithm to the case of hypergraphs. It utilizes hyperedges only for verification. The rule for adding a new vertex is to verify whether there are corresponding hyperedges between it and the matched vertices. [86] utilises the index-filter-verification framework in unlabeled hypergraphs to support similarity subhypergraph matching. [68] works on subhypergraph matching in unlabeled hypergraphs. Since most hypergraphs in real-world applications are labelled, we only focus on exact subhypergraph matching in labelled hypergraphs in this paper. [30] proposes an incident hyperedge structure (IHS) filter for candidate vertex filtering to reduce the number of candidates as we will introduce in the next section. In addition, none of the existing subhypergraph matching algorithms supports parallel execution, which significantly limits their performance in large hypergraphs.

Hypergraph Databases. A hypergraph database is essential for building AGI [28] systems and highly complex, large-scale knowledge representation (KR) systems. AtomSpace [60] is an in-memory KR database for the Open Cognition (OpenCog) project [61]. It adopts a generalised hypergraph (i.e., metagraph) model [27] and features a query engine as well as a rule-driven inferencing engine to perform reasoning. HypergraphDB [42] is an embedded and transactional graph database based on hypergraphs. However, the support of hypergraph pattern queries (i.e., subhypergraph matching) is still on-progress. TypeDB [76] (formerly known as GRAKN [58]) is a strongly-typed database with a rich and logical type system. It also embraces the hypergraph data model, which allows the users to model their domain based on logical and object-oriented principles.

III. BACKGROUND

In this section, we present the formal problem definition and how to extend generic subgraph matching framework to subhypergraph matching.

A. Problem Definition

We focus on undirected, connected and vertex-labelled[2] simple hypergraphs in this paper.

Definition III.1. (Hypergraph). A hypergraph H is defined as a tuple $H = (V, E, l, \Sigma)$ where V is a finite set of vertices and $E \subseteq \mathcal{P}(V) \setminus \{\emptyset\}$ is a set of non-empty subsets of V called hyperedges, where $\mathcal{P}(V)$ is the power set of V. Σ is the set of labels, and l is a label function that assigns each vertex v a label in Σ, denoted as $l_G(v)$ or $l(v)$ when the context is clear.

The number of vertices and hyperedges in H is denoted as $|V(H)|$ and $|E(H)|$, respectively. If a vertex $v \in V(H)$ belongs to a hyperedge $e \in E(H)$ then we say that v and e are *incident*. The collection of hyperedges incident to a vertex v is denoted as $he_H(v)$ or $he(v)$ when the context is clear. The *degree* of a vertex v in H, denoted as $d_H(v)$ or $d(v)$ when the

context is clear, is the number of hyperedges that are incident to v, i.e., $|he_H(v)|$. The *arity* of a hyperedge e, denoted by $a(e)$, is the number of vertices in e, i.e., the number of vertices that are incident to e. We denote $he^a(v)$ as the set of incident hyperedges with arity a. The average arity of H is denoted as $\overline{a_H} = \frac{\Sigma_{e \in E(H)} a(e)}{|E(H)|}$, and the maximum arity is denoted as a_{max}. Two vertices are called *adjacent* if some edge contains both of them. Two hyperedges e_1 and e_2 are called *adjacent* if $e_1 \cap e_2 \neq \emptyset$. We use $adj_H(u)$ or simply $adj(u)$ to represent all adjacent vertices a vertex u. Similarly, $adj_H(e)$ or $adj(e)$ represents all adjacent edges of an edge e.

Definition III.2. (Subhypergraph). A subhypergraph H' of $H = (V, E, l, \Sigma)$ is a hypergraph $H' = (V', E', l, \Sigma)$ where $V' \subseteq V$ and $E' \subseteq E$.

Definition III.3. (Subhypergraph Isomorphism). Given a query hypergraph q and a data hypergraph H, q is subhypergraph isomorphism to H if and only if there is an *injective* mapping $f : V(q) \to V(H)$ such that, $\forall u \in V(q), l_q(u) = l_H(f(u))$, and $\forall e_q = \{u_1, u_2, \ldots, u_k\} \in E(q), \exists e_H = \{f(u_1), f(u_2), \ldots, f(u_k)\} \in E(H)$.

We refer to each isomorphic subhypergraph as a *subhypergraph isomorphism embedding* of q in H. In our match-by-hyperedge framework, we simply represent the query hyperedges as $(e_{q_1}, e_{q_2}, \ldots, e_{q_n})$. The matched embedding is therefore denoted as $m = (e_{H_1}, e_{H_2}, \ldots, e_{H_n})$, where $e_{H_i} = \{f(u) : u \in e_{q_i}\}$ for $1 \leq i \leq n$. We use the notion of $f(e_{q_i}) = e_{H_i}$ in our presentation to denote matched hyperedges, and H_m to denote the subgraph in H constructed by all hyperedges in m. We call a subhypergraph q' of the query hypergraph q a *partial query*, and an embedding of q' a *partial embedding*.

Example III.1. For example, consider the query hypergraph q and data hypergraph H in Fig. 1. By representing q as $(\{u_2, u_4\}, \{u_0, u_1, u_2\}, \{u_0, u_1, u_3, u_4\})$, there are two subhypergraph matching embeddings of q in H, which can be denoted as (e_1, e_3, e_5) and (e_2, e_4, e_6), respectively. Given a partial query with one single hyperedge $q' = (\{u_2, u_4\})$, its matched partial embeddings are (e_1) and (e_2).

Problem Statement. Given a query hypergraph q and a data hypergraph H, we study the task of *subhypergraph matching* to efficiently find all subhypergraph isomorphism embeddings of q in H in parallel on a single machine.

B. Extending Subgraph Matching to Hypergraphs

As discussed, converting hypergraphs into bipartite graphs can dramatically increase the graph size and incur large overhead [39]. Therefore, a better approach for subhypergraph matching is to directly extend one of the existing subgraph matching algorithms as in existing works [15], [16], [30]. In this subsection, we briefly present a *generic* framework of extending a subgraph matching algorithm to subhypergraph matching as our baseline. As most subgraph matching algorithms follow the backtracking framework, we select to extend the backtracking-based subgraph matching framework [53], [70] to subhypergraph matching. Thus, our extension can

---

[2]Our techniques can be easily applied to edge-labelled hypergraphs as well by adding additional constraints of hyperedge labels.

Algorithm 1: Generic Subgraph Matching Framework

Input : A query graph q and a data graph G
Output : All subgraph embeddings from q to G

1   $\mathcal{C}, \mathcal{A} \leftarrow$ generate candidate vertex sets for each query vertex and any other auxiliary data structure ;
2   $\varphi \leftarrow$ generate a matching order;
    /* Recursively match query vertices to data vertices using backtracking.   */
3   Enumerate(q, G, $\mathcal{C}$, $\mathcal{A}$, φ, {}, 1) ;
4   **Procedure** Enumerate(q, G, $\mathcal{C}$, $\mathcal{A}$, φ, M, i)**:**
5   **if** $i = |\varphi| + 1$ **then**
6   output the embedding M ;
7   **return** ;
8     $u \leftarrow$ select a query vertex to match given φ and M ;
9   **foreach** $v \in$ GenerateCandidates($\mathcal{C}$, $\mathcal{A}$, M) **do**
10   **if** v *is a valid mapping of* u **then**
11   Add the mapping (u, v) to M ;
12   Enumerate(q, G, $\mathcal{C}$, $\mathcal{A}$, φ, M, $1 + 1$) ;
13   Remove the mapping (u, v) to M ;

---

be orthogonally applied to most existing subgraph matching algorithms, including [8], [9], [21], [31], [35], [66], etc.

Algorithm 1 illustrates the generic subgraph matching framework [53], [70]. In this generic framework, the first step is to generate a candidate vertex set for each query vertex and builds any auxiliary data structures (Line 1). Then, it computes a matching order of the query vertices (Line 2). Finally, it recursively enumerates all results by sequentially mapping each query vertex to data vertices and backtracking when necessary (Line 3). To validate that two vertices $u \in V(q)$ and $v \in V(G)$ can be successfully matching (Line 10), the following constraint is applied in subgraph matching.

Theorem III.1. *(Subgraph Matching Constraint). Given* $u \in V(q)$ *and* $v \in V(G)$ *are two vertices in the query and data graph, the assignment* $f(u) = v$ *is valid if and only if for all previous vertices* $f(u') = v'$ *in the partial embedding such that* $(u, u') \in E(q)$, *there is an edge* $(v, v') \in E(G)$.

To extend the framework to the case of hypergraphs, the above constraint in Line 10 needs to be generalised in the sense that not only compatibility with respect to (w.r.t.) all edges but w.r.t. all *hyperedges* is checked. Therefore, the following new constraint is applied. Clearly, such an extension is straightforward to implement.

Theorem III.2. *(Subhypergraph Matching Constraint). Given* $u \in V(q)$ *and* $v \in V(G)$ *are two vertices in the query and data hypergraph, the assignment* $f(u) = v$ *is valid if and only if for all previous vertices* $f(u') = v'$ *in the partial embedding such that* $\{u' : u' \in M\} + \{u\} \in E(q)$, *there is an edge* $\{v' : v' \in M\} + \{v\} \in E(H)$.

IHS Filter. To further improve the candidate generation (i.e., GenerateCandidates function in Line 9) in the generic matching framework for hypergraphs, we add and implement the incident hyperedge structure (*IHS*) filter proposed in existing work [30] to our extension when generating candidate vertex set in our implementation. In *IHS* filter, specifically, a data vertex v is included in the candidate set of the query vertex u if the following four conditions are met:

- *Degree and label*: $l(u) = l(v)$ and $d(u) \leq d(v)$
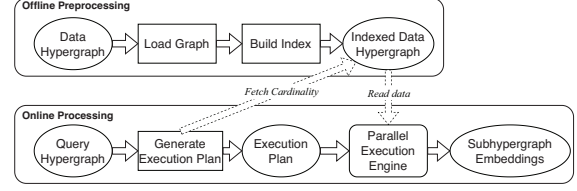- *Number of adjacent nodes*: $|adj(u)| \leq |adj(v)|$

Fig. 3: HGMatch Framework Overview. The solid arrow represents the workflow and the dotted arrow shows the interaction between data and processing steps.

- *Arity containment*: $\forall a, |he^a(u)| \leq |he^a(v)|$
- *Hyperedge labels*: $\exists e_1, e_2, \forall \sigma, |e_1(\sigma)| = |e_2(\sigma)|$, where $e_1 \in he^a(u)$, $e_2 \in he^a(v)$ and $e(\sigma)$ is a set of vertices having label σ in the hyperedge e

Remark. It is worth noting that [30] introduces the *IHS* filter only on the subgraph matching algorithm Turbo$_{ISO}$ [33], whereas in our generic extension the filter can be applied for all backtracking-based subgraph matching algorithm, including more recent algorithms such as CFL [9], DAF [31] and CECI [8]. These algorithms with the *IHS* filter can yield better performance than the original method proposed in [30]. So in our experiments, we use the extended versions of CFL, DAF and CECI as our baseline methods.

IV. HGMatch OVERVIEW

In this section, we introduce the basic workflow of HGMatch followed by the data hypergraph storage mechanism in HGMatch.

A. Overall Workflow

The workflow overview of HGMatch is illustrated in Fig. 3. Specifically, two main stages are *offline data hypergraph preprocessing* and *online query processing*.

In the offline data hypergraph preprocessing stage, the first step is to load the data hypergraph from the source (e.g., text files) and construct the data hypergraph structure (Section IV-B). Once the data hypergraph data structure is constructed, HGMatch builds a lightweight inverted hyperedge index to boost the retrieve process of all incident hyperedges of a given vertex (Section IV-C). At the end of prepossessing, an indexed data hypergraph is created. Note that HGMatch does not build any auxiliary data at runtime, the indexed data hypergraph is created only once offline and it is lightweight, as will be discussed later.

In the online query processing stage, HGMatch receives a query hypergraph as its input. Then, the query hypergraph is sent to the plan generator to generate an execution plan. The plan generator fetches cardinality information for the indexed data hypergraph to select a better matching order. The generated execution plan is then input into HGMatch's parallel execution engine. The execution engine accesses the indexed data hypergraph and executes the given plan to compute all subhypergraph embeddings in parallel.

B. Data Hypergraph Storage

In HGMatch, we store the data hypergraph in multiple *hyperedge tables*, where each hyperedge table has a unique

(a) Partition 1		
$\mathcal{S}(e) = \{A, B\}$		
E	$e_1 = \{v_2, v_4\}$	
	$e_2 = \{v_4, v_6\}$	
I	$v_2 \rightarrow [e_1]$	
	$v_4 \rightarrow [e_1, e_2]$	
	$v_6 \rightarrow [e_2]$	

(b) Partition 2		
$\mathcal{S}(e) = \{A, A, C\}$		
E	$e_3 = \{v_0, v_1, v_2\}$	
	$e_4 = \{v_3, v_5, v_6\}$	
I	$v_0, v_1, v_2 \rightarrow [e_3]$	
	$v_3, v_5, v_6 \rightarrow [e_4]$	

(c) Partition 3		
$\mathcal{S}(e) = \{A, A, B, C\}$		
E	$e_5 = \{v_0, v_1, v_4, v_6\}$	
	$e_6 = \{v_2, v_3, v_4, v_5\}$	
I	$v_0, v_1, v_6 \rightarrow [e_5]$	
	$v_4 \rightarrow [e_5, e_6]$	
	$v_2, v_3, v_5 \rightarrow [e_6]$	

TABLE I: Example Data Layout of the Hypergraph in Fig. 1b. The header of each table represents the signature of all its hyperedges. E represents hyperedges and I represents inverted hyperedge index.

hyperedge signature. We define the concept of hyperedge signature as follows.

Definition IV.1. (Hyperedge Signature). The signature of a hyperedge e, denoted as $\mathcal{S}(e)$, is a *multiset* of all vertex labels contained in e, i.e., $\mathcal{S}(e) = multiset\{l(v) : v \in e\}$.

We denote $he(v, s)$ as the set of incident hyperedges with signature s. HGMatch stores data hyperedges with different hyperedge signatures in separated hyperedge tables denoted as *partitions*. As a result, to search the candidate hyperedges of a query hyperedge e_q, HGMatch only needs to scan the partition with the signature $\mathcal{S}(e_q)$, rather than scanning the whole hypergraph.

Example IV.1. The partitioned hyperedge tables of the data graph H in Fig. 1b are shown in Table I. For the given data hypergraph, HGMatch constructs three partitions having signatures $\{A, B\}$, $\{A, A, C\}$ and $\{A, A, B, C\}$, respectively.

Size Analysis. The proposed hypergraph data structure in HGMatch brings only a very small overhead of an additional signature header for each partition, which is no larger than the size of all hyperedges (i.e., all hyperedges have unique signatures in the worst case). Thus, the total size of storing all hyperedges in HGMatch is $O(\overline{a_H} \times |E(H)|)$.

C. Inverted Hyperedge Index

In subgraph matching in conventional graphs, it is essential to access all connected edges (i.e., neighbours) of a given vertex. In hypergraphs, similarly, it is often demanded to get all incident hyperedges of a given vertex. Given a hyperedge table, it requires a linear scan to complete such an operation which can be time-consuming for large hypergraphs. To further speed up this process of finding all incident hyperedges (with a certain signature) of a given vertex, we adopt the common technique of inverted index [10], [78] to build a lightweight *inverted hyperedge index* for each hyperedge table in HGMatch.

Example IV.2. The inverted hyperedge index of the data graph H in Fig. 1b is also shown in Table I. It maps each vertex in a hyperedge list to a *posting list* of hyperedge IDs of all its incident hyperedges in the hyperedge list (in ascending order).

Size Analysis. The inverted hyperedge index in HGMatch is also lightweight. For each hyperedge, its hyperedge ID will appear in the posting list of all the vertices it contains. There-

---

Algorithm 2: HGMatch Framework

Input : A query hypergraph q and a data hypergraph H
Output : All embeddings from q to H
/* Compute a matching order of query hyperedges. */
1 $\varphi \leftarrow$ ComputeMatchingOrder(q, H) ;
/* Enumerate embeddings of the 1st query hyperedge. */
2 $e_1 \leftarrow$ the 1st query hyperedge in φ ;
3 $M \leftarrow$ all matched hyperedges of e_1 in H ;
/* Iteratively match the remaining query hyperedges. */
4 **foreach** $e_q \in (\varphi - e_1)$ **do**
5    $M' \leftarrow \{\}$;
6   **parallel foreach** $m \in M$ **do**
7      $C(e_q) \leftarrow$ GenerateHyperedgeCandidates(q, H, φ, e_q, m) ;
8   **foreach** $c \in C(e_q)$ **do**
9        $m' \leftarrow m + c$;
10   **if** IsValidEmbedding(q', m') *where q' is the partial query* **then**
11          $M' \leftarrow M' + m'$;
12    $M \leftarrow M'$;
13 **return** M ;

---

fore, each hyperedge e takes additionally $O(a(e))$ space. The total size of the inverted edge index is also $O(\overline{a_H} \times |E(H)|)$.

V. MATCH-BY-HYPEREDGE FRAMEWORK

In HGMatch, we propose a *match-by-hyperedge* framework for efficient subhypergraph matching enumeration. By using hyperedges as the minimal matching sub-structure, HGMatch is able to fully utilise the higher order information in hypergraphs to generate candidates and avoid redundant computation.

The framework is illustrated in Algorithm 2. Given the query hypergraph q and the data hypergraph G, we first compute a *matching order* of the query hyperedges (Line 1), which is defined in Definiton V.1.

Definition V.1. (Matching Order). A matching order in HGMatch, denoted as φ, is a permutation of $E(q)$. $\varphi[i]$ is the ith hyperedge in φ and $\varphi[i : j]$ is the set of hyperedges from index i to j ($1 \leq i \leq j \leq |\varphi|$).

After that, we initialize the set of embeddings M as the hyperedges that match the first query hyperedge in φ, namely $\varphi[0]$ (Line 2-3). Then, for the remaining query hyperedges in φ, HGMatch iteratively expands each partial embedding by one matched hyperedges in parallel until all hyperedges are matched (Line 4-12). When expanding a partial embedding m, we first generate a candidate set $C(e_q)$ for the query hyperedge e_q that is currently being matched (Line 7). Then, a validation is performed to filter out false positives (Line 9).

Note that we present the algorithmic framework of HGMatch (i.e., Algorithm 2) in the form of a bread-first search (BFS) for the ease of presentation, whereas the actual parallel enumeration is scheduled using HGMatch's task-based scheduler (Section VI-B).

A. Matching Order

HGMatch can work with any connected matching order. We compute the matching order of query hyperedges using the cardinality information fetched from the metadata of hyperedge tables to match *infrequent* and *highly connected* hyperedge as early as possible.

Algorithm 3: ComputeMatchingOrder

> **Input** : A query hypergraph q and a data hypergraph H
> **Output** : A matching order φ
> /* Find the hyper with smallest cardinality as a starting hyperedge. */
> **1** $e_1 \leftarrow \arg\min_e Card_y(e, H)$, where $e \in E(q)$;
> **2** $\varphi \leftarrow (e_1)$;
> /* Iteratively add connected hyperedges */
> **3** **while** $|\varphi| \neq |E(q)|$ **do**
> **4** $\quad e_q \leftarrow \arg\min_e \frac{Card(e, H)}{V_\varphi \cap e}$, where $e \in E(q) - \varphi$ and $V_\varphi \cap e \neq \emptyset$;
> **5** $\quad \varphi \leftarrow \varphi + e_q$;
> **6** **return** φ ;

Definition V.2. (Hyperedge Cardinality). The cardinality of a query hyperedge e_q in a data hypergraph H, denoted as $Card(e_q, H)$, is $|\{e : e \in E(H), \mathcal{S}(e) = \mathcal{S}(e_q)\}|$.

The algorithm for computing a matching order is given in Algorithm 3. It starts with the query hyperedge e_1 that has the minimal cardinality in the data graph (Line 1-2). Then, we iterate over all remaining edges in the query graph and add the query hyperedge with small cardinality and a highly connected number of vertices with the existing partial query (Line 4).

Complexity. Since $Card(e, H)$ can be directly accessed from the hyperedge tables (i.e. the number of rows in a table) in $O(1)$, the time complexity of Algorithm 3 is $O(|E(q)|^2)$.

B. Candidates Generation

In this subsection, we introduce how HGMatch generates candidates of a given query hyperedge. We first present some observations used in HGMatch to prune candidates. Then we discuss how HGMatch generates these candidates efficiently using set operations.

Suppose that we are computing the candidates data hyperedge of a query hyperedge $e_q = \varphi[i]$, for a partial embedding m, where the current partial query is q' and the partial query after matching e_q is q. We generate and prune the candidates based on the following observations.

Observation V.1. *(Hyperedge Signature).* The matched hyperedge must have the same hyperedge signature as the query signature, i.e., $\mathcal{S}(f(e_q)) = \mathcal{S}(e_q)$.

Observation V.2. *(Hyperedge Adjacency).* For each adjacent hyperedge e of e_q in $E(q)$, $f(e_q)$ must be adjacent to $f(e)$, i.e., $\forall e \in E(q), e \in adj(e_q), \exists v \in f(e), v \in f(e_q)$.

Observation V.3. *(Hyperedge Non-Adjacency).* For each non-adjacent hyperedge e of e_q in $E(q)$, $f(e_q)$ must not be adjacent to $f(e)$, i.e., $\forall e \in E(q), e \notin adj(e_q), \nexists v \in f(e), v \in f(e_q)$.

Observation V.4. *(Labels and Degree of Incident Vertices).* Suppose $e \in adj(e_q)$, if $u \in e$ and $u \in e_q$, then $\exists v \in f(e_q), v \in f(e), l_H(v) = l_q(u), d_{H_m}(v) = d_{q'}(u)$.

Based on the four observations above, the procedure of candidate generation in HGMatch is given in Algorithm 4. The algorithm firstly obtains a set of vertices in the partial embedding m that must not be incident to the current hyperedge being matched (Line 1). At Line 2, we initialise an empty set of sets of potential hyperedge candidates. Then, for each hyperedge e in the current partial query that is adjacent to

---

Algorithm 4: GenerateHyperedgeCandidates

> **Input** : A query hypergraph q, a data hypergraph H, a matching order φ, a query hyperedge e_q , a partial embedding m
> **Output** : All candidates hyperedges of e_q in H
> /* All vertices in m that are definitely non-incident to $f(e_q)$ */
> **1** $V_{n\_incdt} = \{v : v \in f(e), f(e) \in m, e \in E(q), e \notin adj(q)\}$;
> /* Init an empty set of sets of hyperedges. */
> **2** $C' = \{\}$
> /* Find a set of candidates for each vertex that the matched data hyperedge may be incident to */
> **3** **foreach** $e \in adj(e_q), f(e) \in m$ **do**
> **4** $\quad$ **foreach** $u \in e, u \in e_q$ **do**
> **5** $\quad\quad V_{incdt} \leftarrow \{v : v \in f(e) - V_{n\_incdt}, l_H(v) = l_q(u), d_{H_m}(v) = d_{q'}(u)\}$, where q' is the partial query;
> $\quad\quad$ /* append a set of candidate hyperedges to C' */
> **6** $\quad\quad C' \leftarrow C' + \{\bigcup_{v \in V_{incdt}} he(v, \mathcal{S}(e_q))\}$;
> /* Intersecting all sets of candidates to obtain the final candidates */
> **7** $C(e_q) \leftarrow \bigcap_{E_i \in C'} E_i$;
> **8** **return** $C(e_q)$;

---

e_q, we iterate each query vertex u in e that is also incident to e_q (Line 3-4). For each vertex u, we found a set V_{incdt} of all vertices in $f(e)$ that can be possibly matched to u using Observation V.2, V.3. and V.4 (Line 5). Given V_{incdt}, the candidate hyperedges must be incident to at least one vertex in V_{incdt}, having the same hyperedge signature as e_q (Observation V.1). Therefore, we add the union of hyperedges incident to $v \in V_{incdt}$ with the hyperedge signature of $\mathcal{S}(e_q)$ as an element to C' (Line 6). Lastly, the final candidate set is computed by taking the intersection of all elements (i.e., each element is a set of hyperedges) in C' (Line 7). This is because $f(e_q)$ must be incident to each vertex u in Line 4.

Note that HGMatch employs set difference, union and intersection in candidate generation, which can be implemented very efficiently on modern hardware [32], [41], [54], [78]. With the help of HGMatch's inverted hyperedge index, $he(v, \mathcal{S}(e_q))$ can be quickly fetched in a constant time from the index (Line 6), followed by a direct set intersection (Line 7).

Complexity. In the worst case, assuming that $f(e_q)$ is incident to all previously matched vertices, where each vertex can be contained in $O(E(H))$ hyperedges in the data hypergraph. The time complexity of Algorithm 4 is $O(|V(q)| \times |E(H)|)$.

Example V.1. We give an example for the query and data hypergraph given in Fig. 1. Suppose that the matching order is $\varphi = (\{u_2, u_4\}, \{u_0, u_1, u_2\}, \{u_0, u_1, u_3, u_4\})$ and $m = (e_1 = \{v_2, v_4\}, e_3 = \{v_0, v_1, v_2\})$. To compute $C(\{u_0, u_1, u_3, u_4\})$, the candidate hyperedge must be incident to $v_0, v_1 \in e_3$ and $v_4 \in e_1$. Thus, by accessing the inverted hyperedge index in the partition with signature $s = \mathcal{S}(\{u_0, u_1, u_3, u_4\})$, namely partition 3 in Table I, we compute the candidates as $he(v_0, s) \cap he(v_1, s) \cap he(v_4, s) = \{e_5\}$.

Remark. More filtering rules can be added to HGMatch. However, we found in our experiments that Algorithm 4 yields an extremely low false positive rate, as demonstrated in our experiments. Thus, we do not include more filtering rules to trade off between filtering time and false positive rate.

C. Embedding Validation

The candidates generation in HGMatch may produce false positives. Hence we perform an embedding validation to remove false positives. Suppose the embedding to be verified is m' and the corresponding partial query is q', and we denote the last hyperedge in m' and the corresponding match hyperedge in q' as $e_{m'}$ and $e_{q'}$, respectively. We first use the following observations to validate.

Observation V.5. *(Number of Vertices).* $H_{m'}$ *and* q' *must have the same number of vertices, i.e.,* $|V(H_{m'})| = |V(q')|$.

Obviously, Observation V.5 is trivial to validate. Moreover, to ensure the candidate hyperedge is valid, we can explicitly compute a mapping of vertices between the query and data hyperedge stated in the following lemma.

Lemma V.1. *(Mapping of Vertices). There exist a bijective mapping* $f' : e_{q'} \leftarrow e_{m'}$ *such that,* $\forall u \in e_{q'}, l_q(u) = l_H(f'(u))$, *and* $\forall u \in e, \forall e \in E(q'), f'(u) \in f(e)$.

A naive way to check Lemma V.1 is to compute the vertex mapping using backtracking. However, it introduces many overheads and can be time-consuming similar to the match-by-vertex approach. Instead, we introduce the new concept of *vertex profile* for each vertex in the newly added hyperedge and compare the multisets of vertex profiles for all vertices in the partial query and embedding to verify they are valid.

Definition V.3. *(Vertex Profile.)* A vertex profile of a vertex v in $H_{m'}$ if a tuple $\mathcal{P}(v) = (l(v), he_{H_{m'}}(v))$.

A vertex profile represents the label and all incident hyperedges of a vertex. For a vertex u in a partial query q' and a partial embedding m', we use the notion of $\mathcal{P}_{q' \rightarrow m'}(u) = (l_{q'}(u), \{f(e) : e \in he_{q'}(u)\})$ to denote the profile of its label and the set of all matched hyperedges in the partial embedding of its incident hyperedges. We overload the term $\mathcal{P}(u)$ to denote $\mathcal{P}_{q' \rightarrow m'}(u)$ when context is clear. Using the definition of vertex profile, we have the following theorem.

Theorem V.2. *(Equivalence of Vertex Profile.) Given a partial query* q', *a corresponding partial embedding* m', *with the last hyperedge being match denoted as* $e_{m'}$ *and* $e_{q'}$. *The partial embedding is valid if and only if the two multisets* $\{\mathcal{P}_{q' \rightarrow m'}(u) : u \in e_{q'}\}$ *and* $\{\mathcal{P}(v) : u \in e_{m'}\}$ *are equal.*

Proof. When $\{\mathcal{P}_{q' \rightarrow m'}(u) : u \in e_{q'}\}$ and $\{\mathcal{P}(v) : u \in e_{m'}\}$ are equal, there must exist a bijective mapping that satisfies Lemma V.1. This can be done by mapping each query vertex to a data vertex with the same vertex profile. Since the two multisets are equal, we can definitely construct at least one such bijective mapping. □

Employing the concept of vertex profile, HGMatch avoids backtracking searches. The equivalence of two multisets can be easily compared using hash-based implementation. Combining Observation V.5 and Theorem V.2, HGMatch derives its validation process as illustrated in Algorithm 5.

Example V.2. An example of embedding validation is illustrated in Fig. 4, where we have a partial query q' and a candidate partial embedding m'. Suppose the matching

---

Algorithm 5: IsValidEmbedding

Input : A partial query q', a partial embedding m'
Output : If m' is valid w.r.t. q'
/* Verify the numbers of vertices are equal. */
1 **if** $|V(q')|! = |V(m')|$ **then**
2   **return** False ;
/* Construct the multiset for q'. */
3 $S_{q'} = multiset()$;
4 **foreach** $u \in e_{q'}$ **do**
5     $S_{q'} \leftarrow S_{q'} + \mathcal{P}_{q' \rightarrow m'}(u)$;
/* Construct the multiset for m'. */
6 $S_{m'} = multiset()$;
7 **foreach** $v \in e_{m'}$ **do**
8     $S_{m'} \leftarrow S_{m'} + \mathcal{P}(v)$;
/* Return whether two multisets are equal or not. */
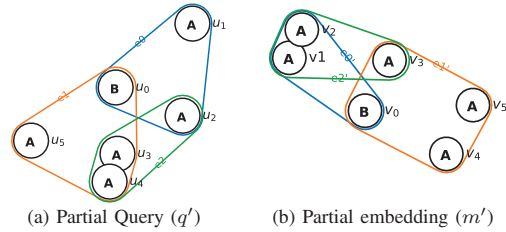9 **return** $S_{q'} = S_{m'}$;

(a) Partial Query (q') (b) Partial embedding (m')

Fig. 4: Example of Embedding Validation.

order of q' is (e_0, e_1, e_2). Currently we have a partial embedding (e'_0, e'_1) of (e_0, e_1) and want to verify the mapping $f(e_2) = e'_2$ is valid. For $e_3 \in q'$, we compute the vertex profile of all its vertices, where $\mathcal{P}(u_2) = (A, \{e'_0, e'_2\})$, $\mathcal{P}(u_3) = (A, \{e'_1, e'_2\})$, and $\mathcal{P}(u_4) = (A, \{e'_1, e'_2\})$. Similarly, for $e'_3 \in m'$, we compute $\mathcal{P}(v_1) = (A, \{e'_0, e'_2\})$, $\mathcal{P}(v_2) = (A, \{e'_0, e'_2\})$, and $\mathcal{P}(v_3) = (A, \{e'_1, e'_2\})$. We notice that $\{\mathcal{P}(u_2), \mathcal{P}(u_3), \mathcal{P}(u_4)\} \neq \{\mathcal{P}(v_1), \mathcal{P}(v_2), \mathcal{P}(v_3)\}$. Hence, m' is not a valid embedding of q'.

Complexity. The time complexity of Algorithm 5 is $O(\overline{a_q} \times |E(q)|)$, as it iterates over every vertex in every query hyperedge.

VI. PARALLEL EXECUTION

In this section, we describe the details of HGMatch's parallel execution engine. On receiving the execution plan of a query hypergraph, HGMatch initialises a *thread pool* of p threads to run the plan in parallel.

A. Dataflow Model

We adopt the popular dataflow model [4], [5], [24], [52], [59], [84] for HGMatch, where computation is abstracted as a dataflow graph [46]. A dataflow graph is a directed acyclic graph (DAG), in which each vertex is an operator, and the directed edges represent data flows. An operator consists of a predefined computing instruction, a certain input and output channels. We introduce three primitive operators SCAN, EXPAND and SINK in HGMatch as follows.

SCAN. SCAN (e_q) is always the first operator in HGMatch. It accepts a query hyperedge e_q as its parameter, iterates over a hyperedge partition of the data hypergraph H, and outputs all data hyperedges with signature $s = \mathcal{S}(e_q)$.
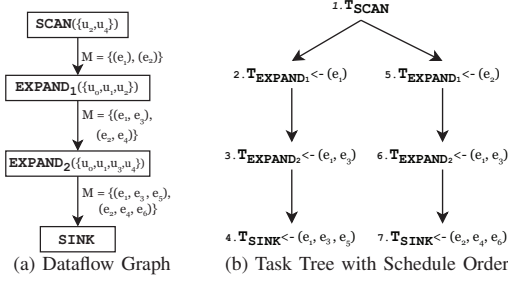
Fig. 5: Example Dataflow Graph and Task Tree with Schedule Order of Fig. 1.

EXPAND. EXPAND (e_q) accepts a query hyperedge e_q as its parameter and partial embeddings as its input. It expands each of its input partial embedding by one more hyperedge e_q (Section V-B and Section V-C), and output new embeddings.

SINK. SINK is always the last operator in HGMatch. It accepts partial embeddings as its input and consumes the results of subhypergraph matching, via either counting or printing all embeddings.

A dataflow graph in HGMatch is always a *direct path* from a SCAN operator to a SINK operator with a number of EXPAND operators in between.

Example VI.1. An example dataflow graph for the query and data hypergraph in Fig. 1 is shown in Fig. 5a, where operators are drawn in rectangles and data flows (i.e., intermediate results) are illustrated next to arrows.

Remark. The dataflow design of HGMatch makes it highly customizable and allows it to be easily extended with other functionalities of hypergraph databases. These can be done by introducing new dataflow operators. Examples include adding extra aggregation and property filtering to the dataflow graph. We leave this as an interesting future work.

B. Task-based Scheduler

A straightforward and commonly used method for parallel execution is to use breath-first search (BFS), which is easy to implement and can achieve high CPU utilisation. However, due to the exponential number of intermediate results of subhypergraph matching, BFS-based scheduling usually consumes a large size of memory and easily results in an out-of-memory error. Depth-first search (DFS), on the other hand, is often adopted for sequential algorithms. DFS is generally hard to parallelise and can lead to low CPU utilisation and load imbalance in the parallel environment. To fulfil the needs of high parallelism degree and low memory consumption at the same time, we design a task-based scheduler in HGMatch to execute the dataflow graph in parallel.

Definition VI.1. (Task). A task is a minimal unit to schedule in HGMatch. It takes either a partition of the data hyperedges or a (partial) embedding as input, executes predefined computation logic, and spawns zero or more new tasks.

There are three types of tasks in HGMatch with respect to the three dataflow operators (i.e., SCAN, EXPAND and SINK). We denote them as T_{SCAN}, T_{EXPAND} and T_{SINK}, respectively.

A T_{SCAN} task takes a partition of the data hyperedges, and spawns a T_{EXPAND} task for each of the hyperedge in the partition. A T_{EXPAND} task takes a partial embedding and spawns a T_{EXPAND} or T_{SINK} task for each expanded partial embedding. A T_{SINK} task takes a partial embedding and counts/outputs the embedding. Once a dataflow graph is constructed, HGMatch executes the dataflow graph using tasks.

Remark. Tasks in HGMatch are *lightweight*. A task contains only a partial embedding and a pointer to the function defining its execution logic (as defined in its corresponding dataflow operator). Hence, the overhead of spawning tasks is very small.

We then discuss how tasks are scheduled in HGMatch. We first assume the thread pool has only one thread to ease our presentation. To control memory consumption, HGMatch employs a *last-in first-out* (LIFO) strategy when scheduling the tasks, in which each thread has a task queue. Specifically, newly spawned tasks will be put to the head of the task queue, and the most recent task is scheduled first (i.e., from the head). The use of LIFO scheduling order avoids materialising all the intermediate results in the memory at one time to efficiently reduce memory consumption.

Example VI.2. A tree of all tasks spawned by the operators in Fig. 5a is given in Fig. 5b. The scheduling order of each task is individuated by the number before it. As shown in the figure, the scheduling employs a LIFO order.

With the scheduler, HGMatch achieves a bounded memory.

Theorem VI.1. HGMatch *schedules a subhypergraph matching job with the memory bound of* $O(\overline{a_q} \times |E(q)|^2 \times |E(H)|)$.

Proof. A dataflow graph in HGMatch has $|E(q)|$ operators. As HGMatch schedules in a LIFO order, in each step, each operator can spawn $|C(e_q)|$ tasks (i.e., $O(|E(H)|)$) to the task queue, where each task takes a partial embedding with at most $|E(q)|$ hyperedges. Representing the partial embedding requires a space of $O(\overline{a_q} \times |E(q)|)$ to maintain all vertex IDs. Therefore, a task queue takes $O(\overline{a_q} \times |E(q)|^2 \times |E(H))$ space. Since there are p task queues in a HGMatch thread pool where p is a pre-configured constant. The overall memory bound is $O(\overline{a_q} \times |E(q)|^2 \times |E(H)|)$. □

C. Load Balancing

Graph computation is usually irregular due to the power-law characteristics in real-world graphs [19], [20]. A simple solution to distribute the load among multiple threads is to assign each thread with an equal share of the firstly matched hyperedges. This *static* and *coarse-grained* technique can still suffer from load skew due to the unpredictable number of embeddings generated by one hyperedge [84]. In HGMatch. we adopt the *fine-grained* work-stealing technique [11], [12], [29], [47] to *dynamically* balance the load.

In HGMatch's dynamic work stealing, each thread in the thread pool maintains a *deque* as its task queue. Since tasks are scheduled in LIFO order new tasks will be spawned into the head of the task queue. Each task execution will pop the most recent task from the head of the task queue. Once a thread has completed its own job by emptying its task queue, it will

TABLE II: Table of Datasets

Dataset	$|V|$	$|E|$	$|\Sigma|$	a_{max}	$\bar{a}$	$|Index|$
HC	1,290	331	2	81	34.8	178KB
MA	73,851	5,444	1,456	1,784	24.2	2.1MB
CH	327	7,818	9	5	2.3	109KB
CP	242	12,704	11	5	2.4	190KB
SB	294	20,584	2	99	8.0	2.1MB
HB	1,494	52,960	2	399	20.5	15.5MB
WT	88,860	65,507	11	25	6.6	6.8MB
TC	172,738	212,483	160	85	4.1	7.8MB
SA	15,211,989	1,103,193	56,502	61,315	23.7	419.7MB
AR	2,268,264	4,239,108	29	9,350	17.1	998.6MB

randomly pick one of the other threads with a non-empty task queue and steal half of the tasks from the tail to balance the load. We implemented a non-blocking deque [17] to reduce the overhead of lock operations. The stealing operates on the minimal scheduling unit in HGMatch, namely task, to achieve near-perfect load balancing in fine grain.

VII. EXPERIMENTS

A. Experimental Setup

HGMatch is implemented in Rust. All experiments are conducted on a server with two 20-core Xeon E5-2698 v4 CPUs (40 threads each) and 512GB memory.

Baselines. We compare HGMatch with the state-of-the-art subgraph matching algorithms CFL [9], DAF [31] and CECI [8]. We adopt the C++ implementations in a recent experimental study of subgraph matching[3] [70], and extend them to the case of hypergraphs as described in Section III-B with additional *IHS* filter. Note that this implementation utilises single instruction multiple data (SIMD) instructions [54] to speed up set intersections. We did not implement SIMD set intersections in HGMatch. The modified algorithms are denoted as CFL-H, DAF-H and CECI-H, respectively. We do not include [30] for the reason discussed at the end of Section III-B. Also, we do not compare [15], [16] since the subgraph matching algorithm they extend, namely the Ullmann's algorithm [75], has been largely outperformed by CFL, CECI and DAF in the literature. We also compare RapidMatch[4] [71]. But since RapidMatch uses join-based techniques which cannot be fitted in our generic backtracking framework, we directly convert the query and data hypergraph to bipartite graphs in RapidMatch.

Datasets. We use 10 real-world data hypergraphs with labelled vertices in our experiment downloaded from [7]. They are house committees (HC), MathOverflow answers (MA), contact high school (CH), contact primary school (CP), senate bills (SB), house bills (HB), Walmart trips (WT), Trivago clicks (TC), StackOverflow answers (SA), and Amazon reviews (AR). We preprocess the datasets to remove all repeated hyperedges and all repeated vertices in one hyperedge. The statistics of the datasets are shown in Table II.

Queries. We use randomly sampled subhypergraphs from the data hypergraphs as our queries. Therefore, for each query hypergraph, there must exist at least one embedding in the corresponding data hypergraph. Specifically, we perform a random walk in the data hypergraph to generate subhypergraphs with the given number of hyperedges whose number

[3]https://github.com/RapidsAtHKUST/SubgraphMatching
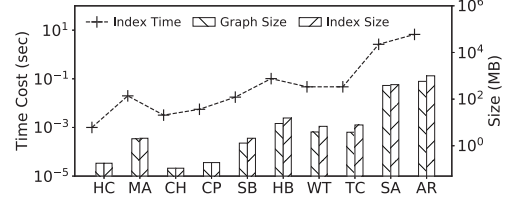[4]https://github.com/RapidsAtHKUST/RapidMatch

Fig. 6: Building Time and Size of Index.

of vertices is in the range of $[|V|_{min}, |V|_{max}]$. The settings of our queries are presented in Table III. We generate 20 random queries for each setting. The random queries vary from low to high selectivity, detailed statistics of all queries are presented in the full version of this paper [85].

Metrics. We measure the average elapsed time of each query type. Each query is executed three times for more precise measurement. We count the number of embeddings for all compared methods instead of outputting them to eliminate I/O costs. Since CFL-H, DAF-H and CECI-H fail on almost all queries on our largest dataset, AR, we only use it for parallel evaluation of HGMatch (Section VII-C). For other datasets, we use them in single-thread evaluation (Section VII-B). For single-thread comparisons (Section VII-B), we set a timeout of 1 hour for all queries. The running time of out-of-time queries will be counted as 3600 seconds when computing the average.

B. Single-thread Comparisons

In this subsection, we evaluate HGMatch in a single-thread environments We use all the datasets except AR as data hypergraphs as discussed before.

Exp-1: Index Building. We first evaluate the proposed inverted hyperedge index. Fig. 6 shows the time of building the index, the size of the graph and the size of the index (in MB). The time of index building is extremely fast. It takes only around 6.7 seconds to build the index even for the largest dataset AR. As for index size, we observe that the index size is similar to the original graph size, which confirms our size analysis in Section IV-C.

Exp-2: Overall Comparisons. We then compare HGMatch with CFL-H, CECI-H, DAF-H, and RapidMatch to verify the efficiency of our matching by hyperedges framework (Section V). The results are shown in Fig. 7. As demonstrated, HGMatch significantly outperforms CFL-H, DAF-H, CECI-H, and RapidMatch in all cases, with an average speedup of 5×10^4, 1×10^5 and 7×10^5, and 1×10^6 respectively. Especially for data hypergraphs with high average arity $\bar{a}$ including HC, MA, HB, and SA, HGMatch outperforms CFL-H, DAF-H and CECI-H by up to 6 orders of magnitude and RapidMatch by up to 7 orders of magnitude. This is because HGMatch can fully use the high-order information in hypergraphs to filter out unpromising hyperedges and reduce redundant computation.

In addition, HGMatch is the only algorithm that completes all queries within the time limit. The query completion rate is shown in Table IV. All algorithms run successfully for smaller datasets (i.e., HC, MA, CH, and CP). However, as the size of data hypergraphs grows, RapidMatch, CFL-H, DAF-H, and

TABLE III: Table of Query Settings

Query	$|E|$	$|V|_{min}$	$|V|_{max}$
q_2	2	5	15
q_3	3	10	20
q_4	4	10	30
q_6	6	15	35

TABLE IV: Query Completion Ratio (Single-thread)

Algorithm	HC	MA	CH	CP	SB	HB	WT	TC	SA	Total
CFL-H					56%	44%	76%	90%	99%	85%
DAF-H		100%			49%	43%	75%	90%	99%	84%
CECI-H					50%	43%	75%	90%	99%	84%
RapidMatch					45%	44%	75%	86%	99%	83%
HGMatch					100%					

CECI-H start to fail on some queries. This is because of the huge search space they have to explore.

Exp-3: Candidates Filtering. In this experiment, we evaluate the pruning power of HGMatch's candidate generation (Section V-B) and embedding validation process (Section V-C). The results are given in Fig. 8, where we count the number of true embeddings (denoted as 'Embeddings'), the number of candidates after applying the vertex number check in Observation V.5 (denoted as 'Filtered'), and the number of candidates generation using Algorithm 4 (denoted as 'Candidates'). We draw the total number of all candidates of all queries for each data hypergraph. For data hypergraphs MA and SA, there are almost no false positive candidates in the candidate set due to a large number of labels. For other datasets with fewer labels, the candidate generation method can generate more false positive embeddings. However, only after a fast check of the number of vertices in the partial embedding, HGMatch are able to filter out the most majority of unpromising results. We observe that 97% of the filtered results are true positive embeddings. This is because, again, HGMatch can fully utilise the high-order information in hyperedges to prune candidates. The results reveal the pruning power of HGMatch, which finally contributes to the significant speedup of existing algorithms.

C. Parallel Comparisons

In this subsection, we evaluate HGMatch in the multi-thread environment. We use our largest dataset, AR, as the data hypergraph as described and the queries in q_3 as the default queries. Note that the original authors implementation of DAF[5] and CECI[6] support parallel execution. However, we do not compare them in our parallel experiments due to the errors in their code that causes segmentation faults and/or reports wrong numbers of embeddings.

Exp-4: Scalability. We conduct a scalability test of HGMatch by varying the number of threads used for parallel execution. We present the results of 2 random queries from q_3 with a large number of embeddings. We denote the two queries as q_3^1 and q_3^2. Specifically, q_3^1 has about 3.86×10^{10} results and q_3^2 has about 2.53×10^8 results. We vary the number of threads from 1 to 60. The results are shown in Fig. 9. HGMatch demonstrates almost perfect linear scalability when the number of threads is equal or below 20 (i.e., $20\times$ speedup when using 20 thread), thanks to the highly optimised parallel execution engine and dynamic load balancing mechanism. When the number of threads is beyond 20, the speedup factor slightly decreases due to non-uniform memory access (NUMA) and hyper-threading in the CPUs of our machine (i.e., 2 physical

[5]https://github.com/SNUCSE-CTA/DAF
[6]https://github.com/iHeartGraph/ceci-release

CPUs with hyper-threading). In the future, we will investigate NUMA optimisations of HGMatch.

Exp-5: Scheduling. In this experiment, we evaluate the memory usage of HGMatch to test the effectiveness of its task-based scheduler (Section VI-B). We compare HGMatch's task-based scheduler with BFS-style scheduling using 20 threads. The memory usages and the number of embeddings of the 20 random queries in q_3 are illustrated in Fig. 10. As the number of embeddings increases, memory usage grows rapidly for BFS-style scheduling. The results indicate for queries with many results, the memory usage of BFS-style scheduling is significantly larger than HGMatch's task-based scheduler because of the materialisation of all intermediate results. This can lead to out-of-memory errors in machines with smaller memory capacities or when querying complex queries. However, HGMatch's task-based scheduler keeps the memory usage bounded with stable memory consumption of around 4.8GB for all 20 queries while achieving almost linear scalability, as demonstrated in the previous experiment.

Exp-6: Load Balancing. We further evaluate the effectiveness of HGMatch's dynamic work stealing mechanism. Due to the space limit, we present the results of q_3^2 in Exp-4 executed using 20 threads. The running time of each worker (i.e.thread) is shown in Fig. 11. Time is reordered to sort in ascending order for ease of illustration. We compare HGMatch with the load balancing technique of assigning the load by the firstly matched hyperedges (denoted as 'HGMatch-NOSTL' in the figure). However, when dynamic work stealing is not applied, we observe load differences among different workers, especially for the last worker. On the other hand, when dynamic work stealing is applied, HGMatch achieves a near-perfect load balancing (the dashed line) with little overhead.

D. Case Study

We demonstrate a case study of subhypergraph matching on question answering over hypergraph knowledge base to help illustrate its applications. We have conducted the case study on the JF17K hypergraph knowledge base dataset [82], which is a small subset of non-binary relations extracted from the knowledge base Freebase [13]. The dataset is a hypergraph with the label for each vertex representing its type. For example, the hyperedges with labels *(Player, Team, Match)* indicate the fact that a football player played in a match representing a team. Another example of hyperedges with labels *(Actor, Character, TV Show, Season)* means an actor played a character in the TV show on the season. We present two example queries in Fig. 12 to answer real-world questions over the knowledge hypergraph by users with respect to the above example hyperedge types. Query 1 (Fig. 12a) is to find the results of 'Football players who represented different teams in different matches'. HGMatch finds 111 embeddings
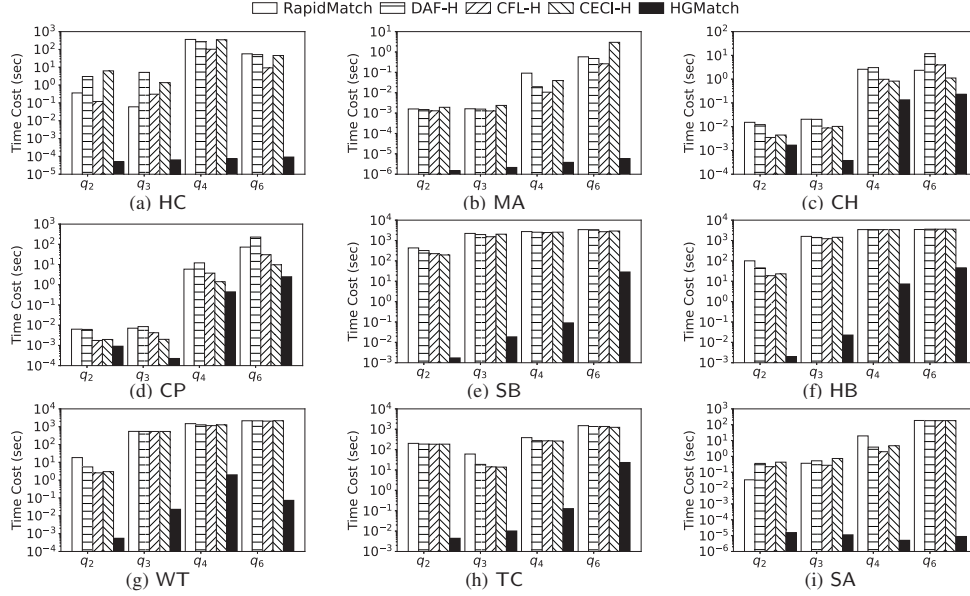
2073

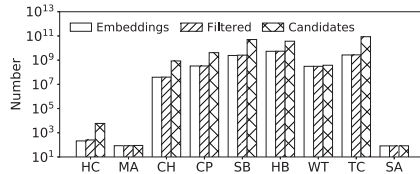Fig. 7: Single-thread Comparisons.

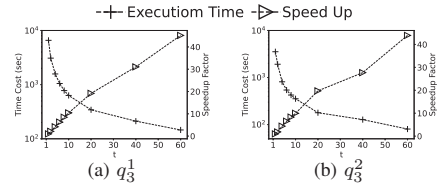Fig. 8: Candidates Filtering.

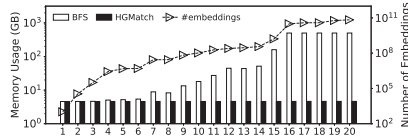Fig. 9: Vary Number of Threads.

Fig. 10: Task-based Scheduling.
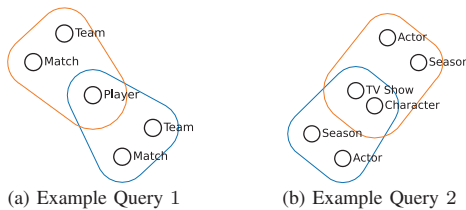

Fig. 11: Work Stealing.

Fig. 12: Example Queries in Hypergraph Knowledge Base.

of this query in the dataset. For instance, the football player Óscar Cardozo played for the Paraguay national football team in FIFA World Cup 2010, but played for the S.L. Benfica team in UEFA Europa League 2014. Query 2 (Fig. 12b) is to find the results of 'Actors who played the same character in a TV show on different seasons'. HGMatch finds 76 embeddings of this query in the dataset. For instance, the actor Carlo Bonomi played the character Pingg in the TV show Pingu during seasons 1-4, and the same character was played by David Sant during seasons 5-6.

VIII. CONCLUSION

We present HGMatch, an efficient parallel system for sub-hypergraph matching. Observed that the existing approaches with the match-by-vertex framework delay hyperedge verification and underutilise high-order information, we propose a novel match-by-hyperedge framework. We store the data hypergraph in tables with a lightweight inverted hyperedge index and use set operations to efficiently generate hyperedge candidates. A highly optimised parallel engine is developed, which adopts the dataflow model. It features a task-based scheduler for bounded-memory execution and a dynamic work-stealing mechanism for load balancing. Experiments show that HGMatch significantly outperforms the baseline algorithms by orders of magnitude on average when using only a single thread and achieves almost linear scalability with near-perfect load balancing when the number of threads increases.

ACKNOWLEDGMENT

REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 431–446, New York, NY, USA, 2016. ACM.

[2] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 62–73. IEEE, 2013.

[3] S. Agarwal, K. Branson, and S. Belongie. Higher order learning with graphs. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 17–24, New York, NY, USA, 2006. Association for Computing Machinery.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[5] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, Feb. 2018.

[6] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[7] A. R. Benson. Hypergraph datasets. https://www.cs.cornell.edu/~arb/data/, 2022.

[8] B. Bhattarai, H. Liu, and H. H. Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1447–1462, New York, NY, USA, 2019. Association for Computing Machinery.

[9] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[10] T. A. Bjørklund, N. Grimsmo, J. Gehrke, and O. Torbjørnsen. Inverted indexes vs. bitmap indexes in decision support systems. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, page 1509–1512, New York, NY, USA, 2009. Association for Computing Machinery.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[13] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1247–1250, New York, NY, USA, 2008. Association for Computing Machinery.

[14] A. Bretto, H. Cherifi, and D. Aboutajdine. Hypergraph imaging: an overview. *Pattern Recognition*, 35(3):651–658, 2002. Image/Video Communication.

[15] H. Bunke, P. Dickinson, and M. Kraetzl. Theoretical and algorithmic framework for hypergraph matching. In *Proceedings of the 13th International Conference on Image Analysis and Processing*, ICIAP'05, page 463–470, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] H. Bunke, P. Dickinson, M. Kraetzl, M. Neuhaus, and M. Stettler. *Matching of Hypergraphs — Algorithms, Applications, and Experiments*, pages 131–154. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[17] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, page 21–28, New York, NY, USA, 2005. Association for Computing Machinery.

[18] L. Chen, Y. Gao, Y. Zhang, S. Wang, and B. Zheng. Scalable hypergraph-based image retrieval and tagging system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 257–268, 2018.

[19] F. Chung, L. Lu, and V. Vu. Spectra of random graphs with given expected degrees. *Proceedings of the National Academy of Sciences*, 100(11):6313–6318, 2003.

[20] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[21] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[22] B. Fatemi, P. Taslakian, D. Vazquez, and D. Poole. Knowledge hypergraphs: Prediction beyond binary relations. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 2191–2197. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[23] S. Feng, E. Heath, B. Jefferson, C. Joslyn, H. Kvinge, H. D. Mitchell, B. Praggastis, A. J. Eisfeld, A. C. Sims, L. B. Thackray, S. Fan, K. B. Walters, P. J. Halfmann, D. Westhoff-Smith, Q. Tan, V. D. Menachery, T. P. Sheahan, A. S. Cockrell, J. F. Kocher, K. G. Stratton, N. C. Heller, L. M. Bramer, M. S. Diamond, R. S. Baric, K. M. Waters, Y. Kawaoka, J. E. McDermott, and E. Purvine. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. *BMC Bioinformatics*, 22(1):287, May 2021.

[24] Flink. Apache flink. https://flink.apache.org/, 2020.

[25] Y. Gao, Z. Zhang, H. Lin, X. Zhao, S. Du, and C. Zou. Hypergraph learning: Methods and practices. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 44(05):2548–2566, may 2022.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[27] D. Gaur, A. Shastri, and R. Biswas. Metagraph: A new model of data structure. In *2008 International Conference on Computer Science and Information Technology*, pages 729–733, 2008.

[28] B. Goertzel. Patterns, hypergraphs and embodied general intelligence. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 451–458, 2006.

[29] A. Grama and V. Kumar. *Load Balancing for Parallel Optimization Techniques*, pages 1905–1911. Springer US, Boston, MA, 2009.

[30] T. W. Ha, J. H. Seo, and M. H. Kim. Efficient searching of subhypergraph isomorphism in hypergraph databases. In *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 739–742, 2018.

[31] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1429–1446, New York, NY, USA, 2019. Association for Computing Machinery.

[32] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1587–1602, New York, NY, USA, 2018. Association for Computing Machinery.

[33] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery.

[34] Y. Han, B. Zhou, J. Pei, and Y. Jia. *Understanding Importance of Collaborations in Co-authorship Networks: A Supportiveness Analysis Approach*, pages 1112–1123. Society for Industrial and Applied Mathematics, 2009.

[35] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 405–418, New York, NY, USA, 2008. Association for Computing Machinery.

[36] W. Hodges et al. *A shorter model theory*. Cambridge university press, 1997.

[37] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao. Answering natural language questions by subgraph matching over knowledge graphs. *IEEE Transactions on Knowledge and Data Engineering*, 30(5):824–837, 2018.

[38] T. Hu, H. Xiong, W. Zhou, S. Y. Sung, and H. Luo. 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*, SIGIR '08, page 871–872, New York, NY, USA, 2008. Association for Computing Machinery.

[39] J. Huang, R. Zhang, and J. X. Yu. Scalable hypergraph learning and processing. In *2015 IEEE International Conference on Data Mining*, pages 775–780, 2015.

[40] T. Hwang, Z. Tian, R. Kuangy, and J.-P. Kocher. Learning on weighted hypergraphs to integrate protein interactions and gene expressions for cancer outcome prediction. In *2008 Eighth IEEE International Conference on Data Mining*, pages 293–302, 2008.

2075

[41] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. VLDB Endow.*, 8(3):293–304, nov 2014.

[42] B. Iordanov. Hypergraphdb: A generalized graph database. In *Proceedings of the 2010 International Conference on Web-Age Information Management*, WAIM'10, page 25–36, Berlin, Heidelberg, 2010. Springer-Verlag.

[43] P. Jian, K. Chen, and C. Zhang. A hypergraph-based context-sensitive representation technique for vhr remote-sensing image change detection. *International Journal of Remote Sensing*, 37(8):1814–1825, 2016.

[44] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng. Fast: Fpga-based subgraph matching on massive graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021.

[45] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1695–1698, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Kavi, Buckles, and Bhat. A formal definition of data flow graph models. *IEEE Transactions on Computers*, C-35(11):940–948, 1986.

[47] R. Kimmig, H. Meyerhenke, and D. Strash. Shared memory parallel subgraph enumeration. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 519–529, 2017.

[48] S. Klamt, U.-U. Haus, and F. Theis. Hypergraphs and cellular networks. *PLOS Computational Biology*, 5(5):1–6, 05 2009.

[49] D. Klein and C. D. Manning. *Parsing and Hypergraphs*, pages 351–372. Springer Netherlands, Dordrecht, 2004.

[50] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *Proc. VLDB Endow.*, 8(10):974–985, June 2015.

[51] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.*, 10(3):217–228, Nov. 2016.

[52] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, Y. Zhang, Z. Qian, and J. Zhou. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.*, 12(10):1099–1112, June 2019.

[53] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.*, 6(2):133–144, Dec. 2012.

[54] D. Lemire, L. Boytsov, and N. Kurz. Simd compression and the intersection of sorted integers. *Softw. Pract. Exper.*, 46(6):723–749, jun 2016.

[55] X. Li, Y. Li, C. Shen, A. Dick, and A. V. D. Hengel. Contextual hypergraph modeling for salient object detection. In *2013 IEEE International Conference on Computer Vision*, pages 3328–3335, 2013.

[56] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[57] T. Menezes and C. Roth. Semantic hypergraphs, 2019.

[58] A. Messina, H. Pribadi, J. Stichbury, M. Bucci, S. Klarman, and A. Urso. Biograkn: A knowledge graph-based semantic database for biomedical sciences. In L. Barolli and O. Terzo, editors, *Complex, Intelligent, and Software Intensive Systems*, pages 299–309, Cham, 2018. Springer International Publishing.

[59] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, July 2019.

[60] OpenCog Foundation. Atomspace. https://wiki.opencog.org/w/AtomSpace, 2022.

[61] OpenCog Foundation. The open cognition project. https://wiki.opencog.org/w/The_Open_Cognition_Project, 2022.

[62] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: On compression and computation. *Proc. VLDB Endow.*, 11(2):176–188, Oct. 2017.

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

[64] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee. Pgx.iso: Parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[65] X. Ren, J. Wang, W.-S. Han, and J. X. Yu. Fast and robust distributed subgraph enumeration. *Proc. VLDB Endow.*, 12(11):1344–1356, July 2019.

[66] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, Aug. 2008.

[67] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 625–636, New York, NY, USA, 2014. Association for Computing Machinery.

[68] Y. Su, Y. Gu, Z. Wang, Y. Zhang, J. Qin, and G. Yu. Efficient sub-hypergraph matching based on hyperedge features. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2022.

[69] S. Sun and Q. Luo. Parallelizing recursive backtracking based subgraph matching on a single machine. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–9, 2018.

[70] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1083–1098, New York, NY, USA, 2020. Association for Computing Machinery.

[71] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He. Rapidmatch: A holistic approach to subgraph query processing. *Proc. VLDB Endow.*, 14(2):176–188, oct 2020.

[72] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

[73] S. Tan, Z. Guan, D. Cai, X. Qin, J. Bu, and C. Chen. Mapping users across networks by manifold alignment on hypergraph. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, page 159–165. AAAI Press, 2014.

[74] L. Tran. Hypergraph and protein function prediction with gene expression data, 2012.

[75] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.

[76] Vaticle Ltd. Typedb. https://github.com/vaticle/typedb, 2022.

[77] L. Vepštas. Graphs, metagraphs, ram, cpu. https://github.com/opencog/atomspace/blob/master/opencog/sheaf/docs/ram-cpu.pdf, 2022.

[78] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 993–1008, New York, NY, USA, 2017. Association for Computing Machinery.

[79] Y. Wang, P. Li, and C. Yao. Hypergraph canonical correlation analysis for multi-label classification. *Signal Processing*, 105:258–267, 2014.

[80] Z. Wang, R. Gu, W. Hu, C. Yuan, and Y. Huang. Benu: Distributed subgraph enumeration with backtracking-based framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 136–147. IEEE, 2019.

[81] J. L. Watkin. An introduction to the clips programming language. 2017.

[82] J. Wen, J. Li, Y. Mao, S. Chen, and R. Zhang. On the representation and embedding of knowledge bases beyond binary relations. In *International Joint Conference on Artificial Intelligence*, 2016.

[83] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, New York, NY, USA, 2012. Association for Computing Machinery.

[84] Z. Yang, L. Lai, X. Lin, K. Hao, and W. Zhang. *HUGE: An Efficient and Scalable Subgraph Enumeration System*, page 2049–2062. Association for Computing Machinery, New York, NY, USA, 2021.

[85] Z. Yang, W. Zhang, X. Lin, Y. Zhang, and S. Li. Hgmatch: A match-by-hyperedge approach for subgraph matching on hypergraphs. https://arxiv.org/abs/2302.06119, 2023.

[86] X. Yu and T. Korkmaz. Hypergraph querying using structural indexing and layer-related-closure verification. *Knowledge and Information Systems*, 46(3):537–565, Mar 2016.

[87] D. Zhou, J. Huang, and B. Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems*, volume 19. MIT Press, 2006.