Efficient Hypergraph Pattern Matching via Match-and-Clean and Intersection Constraint (Full Version)

Siwoo Song
swsong@theory.snu.ac.kr
Seoul National University
Seoul, South Korea

Wonseok Shin
wonseok.shin@standigm.com
Standigm Inc.
Seoul, South Korea

Kunsoo Park*
kpark@theory.snu.ac.kr
Seoul National University
Seoul, South Korea

Giuseppe F. Italiano
gitaliano@luiss.it
LUISS University
Rome, Italy

Zhengyi Yang
zhengyi.yang@unsw.edu.au
University of New South Wales
Sydney, Australia

Wenjie Zhang
wenjie.zhang@unsw.edu.au
University of New South Wales
Sydney, Australia

Abstract

A hypergraph is a generalization of a graph, in which a hyperedge can connect multiple vertices, modeling complex relationships involving multiple vertices simultaneously. Hypergraph pattern matching, which is to find all isomorphic embeddings of a query hypergraph in a data hypergraph, is one of the fundamental problems. In this paper, we present a novel algorithm for hypergraph pattern matching by introducing (1) the intersection constraint, a necessary and sufficient condition for valid embeddings, which significantly speeds up the verification process, (2) the candidate hyperedge space, a data structure that stores potential mappings between hyperedges in the query hypergraph and the data hypergraph, and (3) the Match-and-Clean framework, which interleaves matching and cleaning operations to maintain only compatible candidates in the candidate hyperedge space during backtracking. Experimental results on real-world datasets demonstrate that our algorithm significantly outperforms the state-of-the-art algorithm, by up to three orders of magnitude in terms of query processing time.

PVLDB Reference Format:
Siwoo Song, Wonseok Shin, Kunsoo Park, Giuseppe F. Italiano, Zhengyi Yang, and Wenjie Zhang. Efficient Hypergraph Pattern Matching via Match-and-Clean and Intersection Constraint (Full Version). PVLDB, 18(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitfront.io/r/swsong/ciungWKRzt8K/MaCH/.

1 Introduction

Graphs are widely used to model relationships in various domains, such as social networks, bioinformatics, and chemistry. Traditional graphs represent pairwise relationships between vertices, in which an edge connects exactly two vertices. However, this pairwise representation has limitations when it comes to capturing complex relationships involving multiple vertices simultaneously that are common in many real-world scenarios [15, 17, 26, 27]. A hypergraph is a generalization of a graph by allowing edges to connect any number of vertices. In recent years, hypergraphs have gained increasing attention as they can better represent these complex relationships.

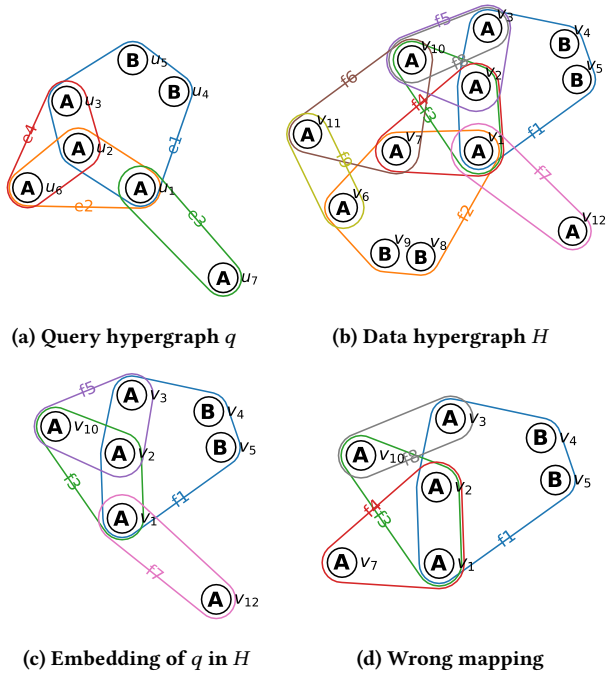


Figure 1: Example of hypergraph pattern matching where 'A' and 'B' are vertex labels, u_i and v_i are vertex IDs, and e_i and f_i are hyperedge IDs.

Pattern matching, which is to find all isomorphic embeddings of a query object in a larger object, is one of the fundamental problems in computer science, and it has been studied in various objects such as strings, trees, graphs, and hypergraphs. If the objects are strings, the pattern matching problem is called string matching [10, 28]; if trees, tree pattern matching [16, 23]; if graphs, subgraph matching [9, 21]; if hypergraphs, hypergraph pattern matching [37] (also

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

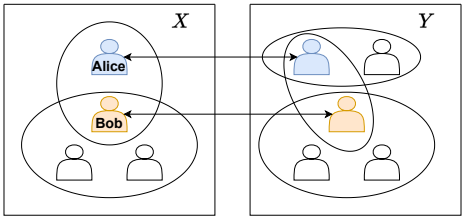


Figure 2: Example of de-anonymizing social networks. Hypergraph pattern matching between an identified network X and an anonymized network Y enables re-identifying users in Y.

known as subhypergraph matching [44]). In this paper we focus on hypergraph pattern matching, which is a fundamental problem in understanding and analyzing hypergraph data. In Figure 1, for example, the subhypergraph in Figure 1c is an embedding (i.e., mapping $\{(e_1, f_1), (e_2, f_3), (e_3, f_7), (e_4, f_5)\}$) of the query hypergraph q (Figure 1a) in the data hypergraph H (Figure 1b).

Hypergraph pattern matching is essential for various applications, such as de-anonymizing social networks [42], analyzing interactions between genes and proteins in biological systems [34], and revealing collaboration trends in the research community [45]. In the context of de-anonymizing social networks, hypergraph pattern matching can be used to re-identify users of an anonymized social network. By modeling a social network as a hypergraph, where a vertex represents a user and a hyperedge represents a friendship or a group membership, a hypergraph pattern matching algorithm can find a mapping of users across networks. As shown in Figure 2, using this mapping information, one can re-identify users in an anonymized social network. However, hypergraph pattern matching is an NP-hard problem [19], requiring substantial computational time for large hypergraph data or queries.

Existing Approaches and Limitations. There are several approaches for hypergraph pattern matching. One approach transforms a hypergraph into a graph by connecting all vertices within each hyperedge as a clique [46]. Subhypergraph embeddings can be found by solving subgraph matching on the transformed graph. However, additional verification is necessary after the subgraph matching, as not all embeddings in the transformed graph correspond to valid embeddings in the original hypergraph. Furthermore, this transform results in a significantly large number of edges in the transformed graph.

Another approach is to transform a hypergraph into a bipartite graph. In this method, vertices and hyperedges from the original hypergraph become vertices in the bipartite graph, with edges representing incidence relationships. Although this approach allows us to apply existing subgraph matching algorithms without additional verification, it fails to utilize the properties specific to hypergraphs, resulting in limited performances [38].

To address such issues, recent works [20, 38, 46] extend an algorithm for subgraph matching to hypergraph pattern matching instead of transforming the hypergraph. Ha et al. [20] extend the subgraph matching algorithm $\text{Turbo}_{\text{ISO}}$ [22], and Yang et al. [44] extend more recent algorithms, namely CFL [9], DAF [21], and CECI [8], to hypergraph pattern matching.

The state-of-the-art method HGMatch [44] introduces the Match-by-Hyperedge framework, which maps query hyperedges directly to data hyperedges instead of mapping query vertices. Additionally, HGMatch proposes a necessary and sufficient condition for checking the validity of embeddings, which doesn't require computing a mapping of vertices. These techniques avoid redundant computation in mapping vertices and significantly improves efficiency.

Despite these advancements, current algorithms still face limitations in efficiently processing queries on massive hypergraphs, particularly when the number of vertices or hyperedges is large.

Contributions. Compared to well-studied subgraph matching, hypergraph pattern matching presents unique challenges due to the complex connectivity patterns in hypergraphs. This complexity significantly increases the difficulty of finding valid embeddings that satisfy all constraints imposed by a query hypergraph. However, these complex connectivity patterns in hypergraphs also provide opportunities for effective pruning of the search space by enabling early detection of invalid embeddings. Thus, strong yet effective constraints for hypergraph pattern matching are crucial for developing efficient algorithms. These constraints can not only prune the search space initially but also continue to prune it as the matching progresses, leading to a more efficient overall matching process.

In this paper, we present a new algorithm, MaCH (Match-and-Clean for Hypergraph Pattern Matching).

(1) We introduce two novel constraints, the *connectivity constraint* for the relationship between two hyperedges and the *intersection constraint* for the relationships involving three or more hyperedges. These constraints effectively capture the complex connectivity patterns in hypergraphs, enabling efficient hypergraph pattern matching. Especially, we prove that the intersection constraint is a necessary and sufficient condition for valid embeddings (Theorem 4.5 and Theorem 6.2). While HGMatch also proposed a necessary and sufficient condition for valid embeddings, called the equivalence of vertex profiles, our condition is faster both theoretically and practically.
Theoretically, for the task of determining whether or not the current partial embedding M can be extended to $M \cup \{(e, f)\}$ for a query hyperedge e and a data hyperedge f, our intersection constraint takes $O(|e|)$ time, where $|e|$ is the arity (i.e., the number of vertices in a hyperedge) of the query hyperedge e, while the equivalence of vertex profiles requires $O(\overline{a_q} \times |E_q|)$ time, where $|E_q|$ is the number of query hyperedges and $\overline{a_q}$ is the average arity (i.e., the average number of vertices per hyperedge) of the query hypergraph. The time complexity of our intersection constraint is an improvement by a factor of $|E_q|$ compared to the equivalence of vertex profiles.
Practically, we compare the verification time spent for checking the intersection constraint in MaCH and that for checking the equivalence of vertex profiles in HGMatch. Our experiment shows that the verification time of MaCH is up to two orders of magnitude faster than that of HGMatch.
Checking the connectivity constraint is lightweight (i.e., takes $O(1)$ time per candidate hyperedge) and effective, and thus it is utilized in both the filtering in (2) below and the cleaning in (3).
(2) We build a new auxiliary data structure, *candidate hyperedge space* (CHS), which stores candidates for query hyperedges

rather than query vertices. CHS better handles vertex automorphisms and the relationships among multiple vertices in hypergraphs, overcoming limitations of vertex-based structures.
HGMatch enumerates embeddings on-the-fly, which may lead to a potentially large search space. CHS allows us to filter out unpromising candidates more effectively. We apply the connectivity constraint to filter out invalid candidates (which cannot be included in any embeddings) in the CHS, significantly reducing the search space. Our experiment shows that up to 99.3% of candidates are removed by the connectivity constraint on the CHS.

(3) We propose a novel framework, *Match-and-Clean*, which interleaves matching and cleaning operations during backtracking. Subgraph matching algorithms typically follow a filtering and matching framework, in which all filtering occurs before the matching process begins. In contrast, our framework integrates cleaning operations directly into the matching process. This framework is particularly well-suited for hypergraph pattern matching in which a hyperedge can connect multiple vertices simultaneously, leading to complex connectivity patterns that are hard to capture in a filtering phase.
HGMatch primarily uses hyperedge signatures (multiset of vertex labels in a hyperedge) to prune candidate hyperedges, leaving some structural information underutilized. We use both the intersection constraint and the connectivity constraint, and eliminate (i.e., clean) candidates in the CHS as the matching progresses. This cleaning significantly prunes the search space, leading to a speedup of up to 20 times in query processing time, when compared to the matching without cleaning.

Experiments on real-world datasets demonstrate that MaCH significantly outperforms HGMatch, the state-of-the-art algorithm for hypergraph pattern matching, by up to three orders of magnitude in terms of query processing time.

2 Preliminaries

2.1 Problem Statement

In this paper, we focus on simple, undirected, and connected hypergraphs with vertices labeled, while the techniques we propose can be readily extended to non-simple, disconnected, hyperedge-labeled, or unlabeled hypergraphs.

Definition 2.1 (Hypergraph). A hypergraph H is defined as a tuple $H = (V, E, L)$ where V is a finite set of vertices and $E \subseteq 2^V$ is a set of non-empty subsets of V called hyperedges, which satisfies $\bigcup_{e \in E} e = V$. L is a label function that assigns each vertex v a label in Σ, which is the set of labels.

In a hypergraph $H = (V_H, E_H, L_H)$, adjacency and incidence are defined as follows. Two vertices $u, v \in V_H$ are adjacent if there exists a hyperedge $e \in E_H$ such that $\{u, v\} \subseteq e$. Two hyperedges $e, f \in E_H$ are adjacent if $e \cap f \neq \emptyset$. A vertex $v \in V_H$ and a hyperedge $e \in E_H$ are incident if $v \in e$. The arity of a hyperedge e, denoted by $|e|$, is the number of vertices in e. The average arity $\overline{a_H}$ of H is defined as $\frac{\sum_{e \in E_H} |e|}{|E_H|}$, and the maximum arity $a_H^{\max}$ is defined as $\max_{e \in E_H} |e|$.

A hypergraph $H' = (V_{H'}, E_{H'}, L_H)$ is a subhypergraph of H if $V_{H'} \subseteq V_H$ and $E_{H'} \subseteq E_H$.

Definition 2.2 (Subhypergraph isomorphism). Given a query hypergraph $q = (V_q, E_q, L_q)$ and a data hypergraph $H = (V_H, E_H, L_H)$, q is subhypergraph isomorphic to H if and only if there is an injective mapping $\phi : V_q \to V_H$ such that, $\forall u \in V_q$, $L_q(u) = L_H(\phi(u))$ and $\forall e_q = \{u_1, \ldots, u_k\} \in E_q$, $\exists e_H = \{\phi(u_1), \ldots, \phi(u_k)\} \in E_H$. We call such an ϕ as subhypergraph isomorphism.

For $e = \{u_1, \ldots, u_k\}$ and subhypergraph isomorphism ϕ, we use the notion $\phi(e) = \{\phi(u_1), \ldots, \phi(u_k)\}$ to denote the mapped hyperedge in the data hypergraph. A *subhypergraph embedding* is represented as the set of hyperedge pairs $\{(e_1, \phi(e_1)), (e_2, \phi(e_2)), \ldots, (e_m, \phi(e_m))\}$ for the set of query hyperedges $E_q = \{e_1, e_2, \ldots, e_m\}$. As in HGMatch [44], embeddings are considered equivalent if their sets of hyperedge pairs are identical, even if the underlying vertex mappings differ. A subhypergraph of the query hypergraph q is called a partial query. An embedding of a partial query in H is called a partial embedding.

In Figure 1, $\{(e_1, f_1), (e_2, f_3), (e_3, f_7), (e_4, f_5)\}$ is an embedding of q in H. One possible underlying subhypergraph isomorphism is $\{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_5), (u_6, v_{10}), (u_7, v_{12})\}$. We can obtain another valid subhypergraph isomorphism by swapping the mappings of u_4 and u_5, resulting in (u_4, v_5) and (u_5, v_4). Despite this change in the vertex mapping, we consider these as the same embedding because the hyperedge mapping remains unchanged.

Problem Statement. Given a query hypergraph q and a data hypergraph H, the *hypergraph pattern matching problem* is to find all subhypergraph embeddings of q in H.

2.2 Related Works

Hypergraph Pattern Matching. Ha et al. [20] extends the subgraph matching algorithm, Turbo$_{\text{ISO}}$ [22], to hypergraph pattern matching, introducing a technique called the incident hyperedge structure (IHS) filter. Their algorithm generates candidates that pass the IHS filter for each query vertex prior to matching and verifies each candidate during the matching phase.

In contrast to previous approaches that map query vertices to data vertices in backtracking, HGMatch [44] introduces the Match-by-Hyperedge framework. This framework directly maps query hyperedges to data hyperedges, thereby reducing redundant computations for enumerating vertex mappings. Instead of generating candidates prior to matching, HGMatch generates candidates for each hyperedge and then verifies partial embeddings extended by these candidates during the matching phase. HGMatch proposes a necessary and sufficient condition for verification, called the equivalence of vertex profiles.

Table 1: Frequently Used Notations

Notation	Definition
q, H	Query hypergraph and data hypergraph
V_h, E_h, L_h	Vertices, hyperedges, labels of a hypergraph h
M	Embedding or partial embedding
$C(e)$	Set of candidate hyperedges for e
$\|e\|$	Arity of a hyperedge e in a hypergraph h
$Sig(\{v_1, v_2, \ldots, v_n\})$	The multiset of labels of vertices v_i's

There are also problems similar to hypergraph pattern matching, with a different definition of an embedding. SubHyMa [38] defines an embedding as a vertex mapping, instead of a hyperedge mapping. It presents several filtering techniques that eliminate invalid candidates for query vertices by utilizing the structural properties of hyperedges and the co-occurrence of vertex pairs within them. HyperISO [46] deals with the problem of subhypergraph containment, in which the vertices in a query hyperedge can be a subset of the vertices in the mapped data hyperedge. It generates hyperedge candidates based on the arity and the number of adjacent hyperedges of a query hyperedge, and it has filtering and matching phases.

Subgraph Matching. Subgraph matching is a special case of hypergraph pattern matching in which an edge connects only two vertices. A subgraph matching algorithm can be used as a subroutine of hypergraph pattern matching or it can be extended to solve hypergraph pattern matching.

Numerous subgraph matching algorithms have been proposed [2, 9, 13, 14, 21, 22, 24, 25, 40]. Recent works have successfully employed the filtering-backtracking approach, utilizing an auxiliary data structure containing a set of candidate vertices for each query vertex. In the filtering phase, these algorithms remove invalid candidates from the data structure. During backtracking, the query vertices are iteratively mapped to one of their candidates according to a matching order. Existing works have developed various pruning techniques to further reduce the search space during backtracking, such as failing sets [21], static and dynamic equivalences [25], vertex and edge no-good [2], pruning by bipartite matching [13], and isolated vertices [24]. Apart from backtracking algorithms, join-based methods [31, 41] for subgraph matching have also been studied, modeling subgraph matching as a join query in a relational database. Interested readers can refer to the extensive surveys of recent subgraph matching algorithms [39, 47].

CSP. A constraint satisfaction problem (CSP) consists of a set of variables, a set of domains which are allowable values for each variable, and a set of constraints that specify allowed combinations of values [35]. The goal is to assign values to all variables such that all constraints are satisfied. Hypergraph pattern matching can be formulated as a CSP by treating the vertices of the query hypergraph as variables, vertices of the data hypergraph as domains, and hyperedges as constraints.

Constraint propagation, a core CSP technique, iteratively applies constraints to eliminate values which violate constraints from domains of variables. Numerous (generalized) arc consistency algorithms have been developed to enhance constraint propagation efficiency, including AC-3 [30], AC-4 [32], AC-6 [5], AC-7 [7], and AC2001/3.1 [6]. The MAC (Maintaining Arc Consistency) algorithms [36] incorporate arc consistency into backtracking search, performing constraint propagation after each variable assignment to prune the search space.

For hypergraph pattern matching, however, this approach can be computationally costly and it yields limited pruning power. The higher-order relationships among multiple vertices represented by hyperedges cannot be fully captured by arc consistency. These relationships necessitate specialized constraints and algorithms for hypergraph pattern matching.

Algorithm 1: MaCH

Input : Query hypergraph q, data hypergraph H
Output: All embeddings of q in H

```
1 C_ini ← BuildCandidateHyperedgeSpace(q, H)
2 Filtering(q, C_ini)
3 M ← ∅
4 MatchAndClean(q, H, C, M)
```

3 Overview

Constraints for Hypergraph Pattern Matching. Unlike graphs with an edge connecting only two vertices, hypergraphs allow a hyperedge to connect multiple vertices, requiring strong yet effective constraints for matching.

In our approach, we introduce two novel constraints, the *connectivity constraint* and the *intersection constraint*. These constraints, along with the hyperedge signature constraint (previously introduced by HGMatch), effectively prune the search space (Section 4).

Building Candidate Hyperedge Space and Filtering. Recent algorithms for subgraph matching use an auxiliary data structure to store candidate data vertices for each query vertex. However, for hypergraph pattern matching, applying such data structures can be inefficient due to two primary factors below.

(1) Vertex automorphisms: Hyperedges often contain multiple vertex automorphisms, leading to redundant computations when matching vertices individually.
(2) Relationships among multiple vertices: A hyperedge connects multiple vertices simultaneously, extending the pairwise connections in traditional graphs. Vertex-based data structures are not suitable for effectively utilizing such relationships in hypergraphs.

To address these limitations, we introduce a *candidate hyperedge space* (CHS), a data structure storing candidates for query hyperedges rather than query vertices. CHS stores potential mappings between hyperedges in the query hypergraph and the data hypergraph. We then apply the connectivity constraint to filter out invalid candidates (which cannot be included in any embeddings) in the CHS, significantly reducing the search space (Section 5).

Match-and-Clean Framework. Although initial filtering is quite effective, it cannot fully capture the complex connectivity patterns of hypergraphs. These patterns, where multiple vertices are connected simultaneously by three or more hyperedges, often become apparent only during the matching process. To address this, we introduce a novel match-and-clean framework, which integrates cleaning operations directly into the matching process. This framework interleaves matching and cleaning operations, utilizing the current partial embedding to clean CHS during the matching process.

(1) We iteratively extend the current partial embedding by mapping a query hyperedge to a candidate data hyperedge in the CHS.
(2) As the new mapping is added, we apply our connectivity and intersection constraints to identify and eliminate candidates which are incompatible with the current partial embedding.

By integrating cleaning operations directly into the matching process, our framework efficiently handles the complex connectivity patterns in hypergraphs (Section 6).

Algorithm 1 shows the outline of our algorithm MaCH, which takes a query hypergraph q and a data hypergraph H as input, and finds all embeddings of q in H.

4 Constraints for Hypergraph Pattern Matching

In this section, we introduce these constraints: Hyperedge signature constraint, Connectivity constraint, and Intersection constraint.

4.1 Hyperedge Signature Constraint

We begin by defining the concept of a signature for a set of vertices, which is fundamental to our constraints.

Definition 4.1. For a set of vertices $S \subseteq V$, the signature of S, denoted as $Sig(S)$, is defined as the multiset of labels of $v \in S$, i.e., $Sig(S) = multiset\{L(v) \mid v \in S\}$.

This definition extends the concept of signature introduced by HGMatch [44], which originally defined it only for hyperedges. Our generalization allows us to apply this concept to arbitrary sets of vertices, which will be crucial for our connectivity constraint and intersection constraint. Using this definition, we can now state the Hyperedge Signature Constraint.

Lemma 4.2 (Hyperedge Signature Constraint). *Given a query hypergraph q and a data hypergraph H, a hyperedge $e \in E_q$ can be mapped to a hyperedge $f \in E_H$ only if e and f have identical signatures.*

4.2 Connectivity Constraint

Matching individual hyperedges based solely on their signatures is not sufficient to ensure a valid partial embedding. To check pairwise relationships between hyperedges, we introduce the concept of connectivity constraint.

Lemma 4.3 (Connectivity Constraint). *Given a query hypergraph q and a data hypergraph H, for any pair of adjacent query hyperedges $e, e' \in E_q$ and data hyperedges $f, f' \in E_H$, a mapping $\{(e, f), (e', f')\}$ can be a partial embedding only if $Sig(e \cap e') = Sig(f \cap f')$.*

Example 4.4. Consider the hypergraphs in Figure 1 and a hyperedge mapping $M = \{(e_1, f_2), (e_2, f_3)\}$. This mapping satisfies the hyperedge signature constraint as $Sig(e_1) = Sig(f_2) = \{A, A, A, B, B\}$ and $Sig(e_2) = Sig(f_3) = \{A, A, A\}$. However, it fails to meet the connectivity constraint because $Sig(e_1 \cap e_2) = \{A, A\}$ while $Sig(f_2 \cap f_3) = \{A\}$. Therefore, M is not a valid partial embedding.

4.3 Intersection Constraint

While the hyperedge signature constraint ensures valid individual hyperedge mappings and the connectivity constraint checks pairwise relationships between hyperedges, these are not sufficient to guarantee a valid embedding as they fail to capture the complex connectivity patterns involving three or more hyperedges.

Theorem 4.5 (Intersection Constraint). *Given a query hypergraph q, a data hypergraph H, and a hyperedge mapping $M : E_q \to E_H$, M is an embedding if and only if $Sig(\bigcap_{e \in S} e) = Sig(\bigcap_{e \in S} M(e))$ for every subset $S \subseteq E_q$.*

Proof. ($\Rightarrow$) Given that M is an embedding, there exists an underlying subhypergraph isomorphism, which is an injective function $\phi : V_q \to V_H$ that preserves vertex labels and satisfies $\phi(e) = M(e)$ for every $e \in E_q$. For any $S \subseteq E_q$, ϕ maps vertices in $\bigcap_{e \in S} e$ to those in $\bigcap_{e \in S} M(e)$, which implies $Sig(\bigcap_{e \in S} e) = Sig(\bigcap_{e \in S} M(e))$.

($\Leftarrow$) We construct a subhypergraph isomorphism ϕ inductively. For each subset $S \subseteq E_q$ of size i (from $|E_q|$ down to 1), we map every unmapped vertex in $\bigcap_{e \in S} e$ to an unmapped vertex in $\bigcap_{e \in S} M(e)$ with the matching label.

For the base case $S = E_q$, since signatures of $\bigcap_{e \in E_q} e$ and $\bigcap_{e \in E_q} M(e)$ are the same, there are the same number of vertices for each label in $\bigcap_{e \in E_q} e$ and $\bigcap_{e \in E_q} M(e)$. Thus, there exists a bijective mapping between them that preserves labels. We define ϕ on the vertices in $\bigcap_{e \in E_q} e$ (as shown in dark blue in Figure 5a) as this bijective mapping.

Now assume that we have constructed ϕ for all subsets S of size greater than k. For a set S of size k, consider the unmapped vertices in $\bigcap_{e \in S} e$ (as shown in blue in Figure 5b) and the unmapped vertices in $\bigcap_{e \in S} M(e)$ (as shown in blue in Figure 5c). Since $Sig(\bigcap_{e \in S} e) = Sig(\bigcap_{e \in S} M(e))$, there are the same number of vertices for each label in $Sig(\bigcap_{e \in S} e)$ and $Sig(\bigcap_{e \in S} M(e))$. For each vertex $u \in \bigcap_{e \in S} e$ which is already mapped, the corresponding vertex $\phi(u) \in \bigcap_{e \in S} M(e)$ is also mapped by the inductive assumption. This one-to-one correspondence between mapped vertices ensures that there are the same number of unmapped vertices for each label in $\bigcap_{e \in S} e$ and $\bigcap_{e \in S} M(e)$, and thus there exists a bijective mapping between unmapped vertices that preserves labels. We define ϕ on the unmapped vertices in $\bigcap_{e \in S} e$ as this bijective mapping.

By induction, ϕ is a subhypergraph isomorphism that satisfies $\phi(e) = M(e)$ for every $e \in E_q$. That is, M is a subhypergraph embedding with the underlying subhypergraph isomorphism ϕ. □

Example 4.6. Consider the hypergraphs in Figure 1 and a hyperedge mapping $M = \{(e_1, f_1), (e_2, f_3), (e_3, f_8), (e_4, f_4)\}$ shown in Figure 1d. This mapping satisfies both the hyperedge signature constraint for each hyperedge and the connectivity constraint for every pair of adjacent hyperedges. For instance, $Sig(e_1 \cap e_2) = Sig(f_1 \cap f_3) = \{A, A\}$. However, $Sig(e_1 \cap e_2 \cap e_4) = \{A\}$ as it includes only one vertex u_2, while $Sig(f_1 \cap f_3 \cap f_4) = \{A, A\}$ as it includes two vertices v_1 and v_2. Thus, M is not a valid embedding.

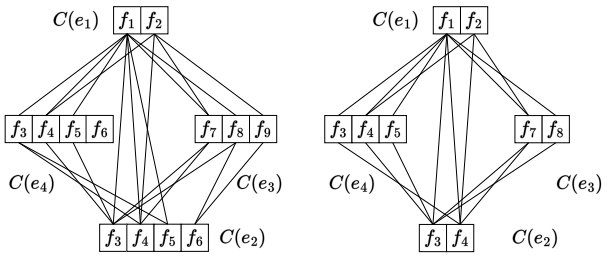
Unlike previous constraints, the intersection constraint provides a necessary and sufficient condition for a mapping to be an embedding. This stronger constraint allows us to precisely characterize valid embeddings. Building on this, we introduce the concept of M-compatibility for candidates.

Definition 4.7. Let M be a partial embedding for a partial query q' of q and let $e \in E_q \setminus E_{q'}$ be an unmapped query hyperedge. A candidate f for e is *M-compatible* if $M \cup \{(e, f)\}$ is a partial embedding for the partial query induced by $E_{q'} \cup \{e\}$.

5 Candidate Hyperedge Space

5.1 Building Candidate Hyperedge Space

Definition 5.1. Given a query hypergraph q and a data hypergraph H, a *Candidate Hyperedge Space (CHS)* on q and H consists of



(a) Initial CHS **(b) Filtered CHS**

Figure 3: Candidate hyperedge space on q and H in Figure 1 before and after filtering

the candidate hyperedge set $C(e)$ for each $e \in E_q$ and connections between the candidates as follows.

(1) For each $e \in E_q$, candidate hyperedge set $C(e)$ is a set of hyperedges in H that e can be mapped to.
(2) There is a connection between $f \in C(e)$ and $f' \in C(e')$ if e and e' are adjacent in q, f and f' are adjacent in H, and $Sig(e \cap e') = Sig(f \cap f')$.

We initialize the CHS as follows. For each query hyperedge e, initial candidate hyperedge set $C_{ini}(e)$ is defined as the set of data hyperedges f which satisfy $Sig(f) = Sig(e)$. We then establish connections between candidate hyperedges. For each pair of adjacent query hyperedges e and e', we create a connection between $f \in C(e)$ and $f' \in C(e')$ if f and f' are adjacent and $Sig(e \cap e') = Sig(f \cap f')$.

Example 5.2. Figure 3a shows the initial CHS C_{ini} for query hypergraph q and data hypergraph H in Figure 1. For e_1, $Sig(e_1) = \{A, A, A, B, B\}$. As f_1 and f_2 have the same signature as e_1, $C_{ini}(e_1) = \{f_1, f_2\}$. Similarly, candidate hyperedge sets for other query hyperedges include data hyperedges with matching signatures.

In q, e_1 and e_2 are adjacent with $Sig(e_1 \cap e_2) = \{A, A\}$. Candidates for e_1 and e_2 have a connection if their intersections match this signature. For instance:

- $f_1 \in C(e_1)$ and $f_3 \in C(e_2)$ are connected as $Sig(f_1 \cap f_3) = \{A, A\}$,
- $f_1 \in C(e_1)$ and $f_6 \in C(e_2)$ are not connected as $f_1 \cap f_6 = \emptyset$,
- $f_2 \in C(e_1)$ and $f_3 \in C(e_2)$ are not connected as $Sig(f_2 \cap f_3) = \{A\}$.

Note that e_4 and e_3 are not adjacent in q, so their candidates do not have connections in the CHS, regardless of their intersections in H.

All hyperedges constituting an embedding are present in the CHS. We call this property of CHS as *complete*.

The number of candidates in CHS is $O(|E_q||E_H|)$, and the number of connections is $O(|E_q|^2|E_H|^2)$ when all hyperedges in the query hypergraph and all hyperedges in the data hypergraph are adjacent to each other, respectively. But, in practice these numbers are much smaller.

We define adjacent candidate set $C(e' \mid e, f)$ as the set of hyperedges in H that e' can be mapped to when e is mapped to f. That is, $C(e' \mid e, f)$ is the set of $f' \in C(e')$ which is connected to $f \in C(e)$.

Algorithm 2: Filtering(q, C)

```
Input : Query hypergraph q and CHS C
Initialize queue Q with all pairs of adjacent query hyperedges
while Q is not empty do
    (e, e') ← Q.pop()
    removed ← false
    foreach f ∈ C(e) do
        if C(e' | e, f) = ∅ then
            Remove f from C(e)
            removed ← true
    if removed then
        foreach e'' adjacent to e do
            if e'' ≠ e' & (e'', e) ∉ Q then
                Q.push((e'', e))
```

5.2 Filtering by Connectivity Constraint

Before the matching phase, we apply the connectivity constraint to the initial CHS as a filtering method to remove candidate hyperedges that cannot be included in an embedding. For a query hyperedge e to be mapped to a candidate f, f must have at least one connected candidate in $C(e')$ for each query hyperedge e' adjacent to e. Formally, $C(e' \mid e, f) \neq \emptyset$ should hold for every e' adjacent to e.

In Algorithm 2, we iteratively remove candidates that fail to meet the connectivity constraint until the CHS contains no candidates that violate the constraint. The algorithm begins by initializing a queue with all pairs of adjacent query hyperedges. Then it iterates through these pairs (e, e') in the queue, examining the candidates for hyperedge e. For each candidate $f \in C(e)$, if f has no connected candidates in $C(e')$ (i.e., $C(e' \mid e, f)$ is empty), f is removed from $C(e)$. When candidates are removed from $C(e)$, the algorithm propagates these changes by adding new query hyperedge pairs (e'', e) to the queue for e'' adjacent to e. The algorithm terminates when the queue is empty, indicating that no further filtering is possible.

This pruning significantly reduces the search space for the subsequent matching phase by decreasing both the number of candidates and the connections between them. The time complexity of Algorithm 2 is described in Appendix A.1.

Example 5.3. Figure 3 shows the initial CHS and the CHS after filtering by connectivity constraint. In the initial CHS, $f_6 \in C(e_4)$ has no connection to candidates of e_1 (and also e_2), so we remove f_6 from $C(e_4)$. Similarly, $f_5 \in (e_2)$ has no connection to candidates of e_3, so we remove f_5 from $C(e_2)$. $f_6 \in C(e_2)$ has no connection to candidates of e_1, so we remove f_6 from $C(e_2)$. After removing f_6 from $C(e_2)$, $f_9 \in C(e_3)$ loses its only connection to e_2's candidates, so we remove f_9 from $C(e_3)$. In the filtered CHS, every remaining candidate has at least one connection to candidates of every adjacent query hyperedge.

6 Match-and-Clean

Algorithm 3 finds subhypergraph embeddings using a dynamic matching order. It extends the partial embedding M to $M \cup \{(e', f')\}$

Algorithm 3: MatchAndClean(q, H, C, M)

Input: Query hypergraph q, data hypergraph H, CHS C, partial embedding M

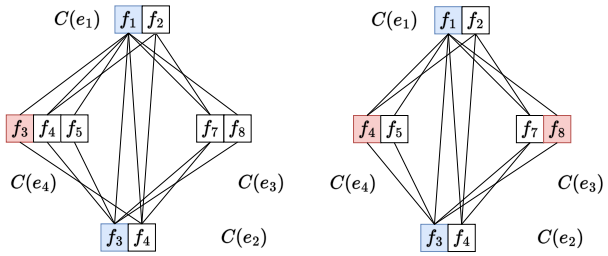
1 **if** $\exists e \in E_q$ *s.t.* $C(e) = \emptyset$ **then**
2 **return** // No valid embedding extended from M
3 $e' \leftarrow$ MatchingOrder(q, C, M)
4 **foreach** $f' \in C(e')$ **do**
5 $M' \leftarrow M \cup \{(e', f')\}$
6 Update b_q and b_H
7 **if** $|M'| = |E_q|$ **then**
8 Report M'
9 **else**
  // Cleaning by connectivity constraint
10 **foreach** *unmapped* $e \in E_q$ *adjacent to* e' **do**
11 **foreach** $f \in C(e)$ **do**
12 **if** $f \in C(e)$ is not connected to $f' \in C(e')$ **then**
13 Remove f from $C(e)$
  // Cleaning by intersection constraint
14 **foreach** *unmapped* $e \in E_q$ **do**
15 **foreach** $f \in C(e)$ **do**
16 CheckIntersection(q, H, C, M', e, f)
17 MatchAndClean(q, H, C, M')
18 Restore b_q and b_H
19 **return**

by mapping a query hyperedge e' to its candidate hyperedge f'. After each extension, it applies a two-stage cleaning process to maintain M-compatibility for all candidates in the candidate hyperedge space.

(1) Connectivity Constraint Check: A fast, coarse-grained cleaning technique that quickly eliminates candidates incompatible with the current partial embedding in $O(1)$ time per candidate (lines 10–13).
(2) Intersection Constraint Check: A more thorough, fine-grained cleaning, ensuring that all remaining candidates are M-compatible. As we will show in Theorem 6.8, this check takes $O(|e|)$ time for each pair of an unmapped query hyperedge e and its candidate $f \in C(e)$ (lines 14–16).

This two-stage process enables early removal of invalid candidates by the quick connectivity constraint check before applying the more expensive intersection constraint check. It achieves improved overall efficiency, when compared to applying only the intersection constraint check, which also obtains the same M-compatibility.

If the candidate set of any query hyperedge becomes empty after the two cleaning stages, the current partial embedding cannot be extended to a valid embedding, and the algorithm backtracks to try the next candidate. Otherwise, the algorithm continues with the extended partial embedding.



(a) Connectivity constraint **(b) Intersection constraint**

Figure 4: Match-and-Clean applied to CHS on q and H in Figure 1. Current partial embedding M is $\{(e_1, f_1), (e_2, f_3)\}$ (blue). Incompatible candidates (red) are cleaned by connectivity constraint in (a) and intersection constraints in (b).

6.1 Cleaning by Connectivity Constraint

Lines 10–13 of Algorithm 3 are to apply the connectivity constraint to the candidates in CHS after a new hyperedge pair (e', f') is added to the partial embedding M.

The algorithm iterates through all unmapped query hyperedges e that are adjacent to the newly mapped hyperedge e'. For each such e, it examines $C(e)$ and removes any candidate f that is not in $C(e \mid e', f')$. For each candidate $f \in C(e)$, it takes $O(1)$ time to check whether $f \in C(e)$ is connected to the newly mapped hyperedge pair (e', f'), as we already built connections in the CHS.

It is important to note the distinction between cleaning by connectivity constraint and filtering by connectivity constraint, which occurs before matching. Filtering removes candidates that don't have connections to any candidates of adjacent query hyperedges in the CHS. In contrast, cleaning removes candidates that don't have connections specifically to the newly mapped hyperedge pair.

Example 6.1. Consider Figure 4a with a partial embedding $M = \{(e_1, f_1), (e_2, f_3)\}$, where (e_2, f_3) is a newly mapped hyperedge pair. We clean the candidate sets of unmapped query hyperedges by the connectivity constraint. f_3 is removed from $C(e_4)$ because there is no connection between the newly mapped pair (e_2, f_3) and the candidate $f_3 \in C(e_4)$. In contrast, f_4 and f_5 remain in $C(e_4)$ as they have connections to $f_3 \in C(e_2)$.

6.2 Cleaning by Intersection Constraint

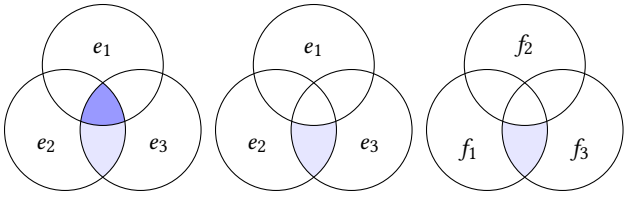
Algorithm 4 is to apply the intersection constraint to the candidate $f \in C(e)$ for the partial embedding M. It removes (i.e., cleans) the candidate from CHS if it is not M-compatible. To check the M-compatibility of a candidate, we use Theorem 4.5, which states that an embedding is valid if and only if, for every subset S of query hyperedges, the signature of the intersection of S matches the signature of the intersection of the corresponding data hyperedges for S. However, computing the signature for the intersection of every subset can take exponential time.

To address this, we shift our focus from intersections to *cells*. Consider a set E of query hyperedges. For a subset S of E, the cell of S is defined as the set of vertices which are only incident to hyperedges in S and not incident to hyperedges in $E \setminus S$. For example, in Figure 5a, the intersection of e_2 and e_3 can be divided into two

Algorithm 4: CheckIntersection(q, H, C, M, e, f)

Input: Query hypergraph q, data hypergraph H, CHS C, partial embedding M, query hyperedge e, data hyperedge f

foreach $u \in e$ **do**
  $b_q[u] \leftarrow b_q[u] + 2^{pos(e)}$
foreach $v \in f$ **do**
  $b_H[v] \leftarrow b_H[v] + 2^{pos(e)}$
$S_q^+ \leftarrow [(b_q[u], L_q[u]) \mid u \in e]$
$S_H^+ \leftarrow [(b_H[v], L_H[v]) \mid v \in f]$
if $sort(S_q^+) \neq sort(S_H^+)$ **then**
  Remove f from $C(e)$
Restore IHBs



(a) Intersection $e_2 \cap e_3$ **(b) Cell of $\{e_2, e_3\}$** **(c) Cell of $\{f_1, f_3\}$**

Figure 5: Intersection and cells of hyperedges for the partial embedding $\{(e_1, f_2), (e_2, f_1), (e_3, f_3)\}$

cells: the cell of $\{e_1, e_2, e_3\}$, representing $e_1 \cap e_2 \cap e_3$ (in dark blue), and the cell of $\{e_2, e_3\}$, representing $e_2 \cap e_3 \setminus e_1$ (in light blue).

A cell can be represented by a bitmap, where each bit position corresponds to a hyperedge in E. In Figure 5, let's assume that $pos(e_1)$ (i.e., bit position of e_1) is 0 (rightmost position), $pos(e_2) = 1$, and $pos(e_3) = 2$. In the bitmap of a cell S, the bit position of a hyperedge e has 1 if and only if e is in S, i.e., the colored cell in Figure 5b is represented as the bitmap 110. (Note that the bitmap is the bit representation of integer $\sum_{e \in S} 2^{pos(e)}$.)

For a query hyperedge set S and its corresponding data hyperedge set $M(S)$ under a partial embedding M, the cell of $M(S)$ is represented by the same bitmap as the cell of S. For example, in Figure 5c, the cell of $\{f_1, f_3\}$ is represented by the bitmap 110, where the mapping is $\{(e_1, f_2), (e_2, f_1), (e_3, f_3)\}$.

Theorem 6.2. *Given a partial query q' of q and hyperedge mapping $M : E_{q'} \rightarrow E_H$, M is a partial embedding if and only if the signature of a cell S matches the signature of $M(S)$ for every subset of query hyperedges $S \in E_{q'}$.*[1]

We introduce two new auxiliary data structures, the *Incident Hyperedge Bitmap* (IHB) and the *Cell Signature* to compute the signatures of cells. These structures enable us to perform M-compatibility checks for each candidate $f \in C(e)$ in $O(|e|)$ time, where $|e|$ is the arity of the query hyperedge e.

Definition 6.3 (Incident Hyperedge Bitmap). For a vertex $u \in V_q$, the Incident Hyperedge Bitmap $b_q^M(u)$ is the bitmap of the cell that u belongs to. Similarly, for a vertex $v \in V_H$, IHB $b_H^M(v)$ is the bitmap of the cell that v belongs to.

Both b_q^M and b_H^M can be computed incrementally when a new mapping (e, f) is added to a partial embedding M.

Example 6.4. Consider the hypergraphs in Figure 1, a hyperedge mapping $M = \{(e_1, f_1), (e_2, f_3), (e_3, f_8), (e_4, f_4)\}$ shown in Figure 1d, and bit positions of e_1, e_2, e_3, e_4 are 0, 1, 2, 3, respectively.

- $b_q^M(u_1) = 0111$ because u_1 is in the cell of $\{e_1, e_2, e_3\}$.
- $b_q^M(u_2) = 1011$ because u_2 is in the cell of $\{e_1, e_2, e_4\}$.
- $b_q^M(u_3) = 1001$ because u_3 is in the cell of $\{e_1, e_4\}$.
- $b_H^M(v_1) = 1011$ because v_1 is in the cell of $\{f_1, f_3, f_4\}$, whose elements are mapped to e_1, e_2, e_4 respectively.
- $b_H^M(v_2) = 1011$ because v_2 is in the cell of $\{f_1, f_3, f_4\}$, whose elements are mapped to e_1, e_2, e_4 respectively.
- $b_H^M(v_3) = 0101$ because v_3 is in the cell of $\{f_1, f_8\}$, whose elements are mapped to e_1, e_3 respectively.

In addition, we introduce Cell Signatures, $\mathcal{I}_q^M(b, l)$ and $\mathcal{I}_H^M(b, l)$. Given a partial query q' of q and hyperedge mapping $M : E_{q'} \rightarrow E_H$, $\mathcal{I}_q^M(b, l)$ (resp. $\mathcal{I}_H^M(b, l)$) stores the signature of the cell in the domain of M (resp. the image of M) represented by bitmap b by counting the number of vertices with label l in the cell.

Example 6.5. Consider the hypergraphs in Figure 1, a hyperedge mapping $M = \{(e_1, f_1), (e_2, f_3), (e_3, f_8), (e_4, f_4)\}$ shown in Figure 1d.

- $\mathcal{I}_q^M(1011, A) = 1$ since u_2 is the only vertex whose IHB is 1011 and label is A.
- $\mathcal{I}_q^M(1001, A) = 1$ since u_3 is the only vertex whose IHB is 1001 and label is A. Although u_2 is also in $e_1 \cap e_4$, u_2 is incident to e_2 as well, which makes its IHB 1011.
- $\mathcal{I}_H^M(1011, A) = 2$ since there are two vertices v_1 and v_2 whose IHB is 1011 and label is A.

For a partial embedding M, to check M-compatibility of a candidate $f \in C(e)$, it is sufficient to compare the signatures of cells by Theorem 6.2. Furthermore, since M is already a valid partial embedding, we only have to check signatures of cells that have changed due to the addition of mapping (e, f). Thus, instead of checking all entries in $\mathcal{I}_q^{M'} = \mathcal{I}_H^{M'}$ for $M' = M \cup \{(e, f)\}$, we only need to consider the signatures of the cells that are subsets of e.

Theorem 6.6. *Let $M : E_{q'} \rightarrow E_H$ be a partial embedding for a partial query q' of q. A candidate $f \in C(e)$ for a query hyperedge $e \in E_q \setminus E_{q'}$ is M-compatible if $\mathcal{I}_q^{M'}(b, l) = \mathcal{I}_H^{M'}(b, l)$ for the bitmap b of every cell S that is a subset of e and every label l.*[1]

Example 6.7. Consider Figure 4b with a partial embedding $M = \{(e_1, f_1), (e_2, f_3)\}$. We clean the candidate sets of unmapped query hyperedges using the intersection constraint. We check for a potential extension $M' = \{(e_1, f_1), (e_2, f_3), (e_4, f_4)\}$. For this extension, we find that $\mathcal{I}_q^{M'}(1011, A) = 1$ while $\mathcal{I}_H^{M'}(1011, A) = 2$. This difference in the cell signatures shows that f_4 cannot be mapped to e_4, so we remove it from $C(e_4)$. Similarly, f_8 is removed from $C(e_3)$. After this cleaning, each of e_3 and e_4 has only one remaining candidate. Mapping these candidates results in the embedding shown in Figure 1c.

[1] Proof is in Appendix.

Algorithm 4 applies the intersection constraint to the candidate $f \in C(e)$, and it removes f if f is not M-compatible. Although there are $2^{|E_q|}$ bitmaps in the definition of the cell signature $\mathcal{I}_q^{M'}$ for mapping $M' = M \cup \{(e, f)\}$, at most $|e|$ entries of the cell signature are non-zero because the hyperedge e has $|e|$ vertices, and each vertex in e can be included in only one cell. Thus, instead of computing the cell signature for every bitmap of a cell that is a subset of e, we create an array of non-zero entries, i.e., $(b_q^{M'}[u], L_q[u])$ for each vertex u in the query hyperedge. Similarly, we create an array of $(b_H^{M'}[v], L_H[v])$ for each vertex v in the data hyperedge, and check whether these two arrays are equal.

Algorithm 4 computes $b_q^{M'}$ and $b_H^{M'}$ by considering each $u \in e$ and each $v \in f$ in lines 1–4. Then, it computes non-empty cells that are subsets of e and f (S_q^+ and S_H^+ in lines 5–6). In lines 7–8, it compares two arrays after sorting these arrays by radix sort, and removes the candidate f if the arrays are not equal. For the next mapping (e, f), Algorithm 4 guarantees that f is M-compatible (i.e., $M \cup \{(e, f)\}$ is a valid partial embedding) by Theorem 6.6.

Theorem 6.8. *The running time of Algorithm 4 is $O(|e|)$.*

Proof. It takes $O(|e|)$ time to compute IHB, to compute non-empty entries, and to sort and compare two arrays. □

HGMatch also presents a necessary and sufficient condition for hypergraph pattern matching called equivalence of vertex profiles. This condition can be used to check whether a candidate $f \in C(e)$ is M-compatible in $O(\overline{a_q} \times |E_q|)$ time, where $\overline{a_q}$ is the average arity of the query hypergraph. The time complexity of our approach is an improvement by a factor of $|E_q|$ compared to HGMatch.

6.3 Matching Order

Line 3 of Algorithm 3 selects a hyperedge based on the following criteria.

(1) Unmapped hyperedges with only one candidate are selected first.
(2) If no such hyperedge exists, we choose the hyperedge with the largest number of unmapped adjacent hyperedges.
(3) In case of a tie, we select the hyperedge with the smallest number of candidates.

We prioritize unmapped hyperedges with only one candidate, because these hyperedges must be mapped eventually, and their early mapping immediately reduce the candidates on other hyperedges. We also give priority to hyperedges with many unmapped adjacent hyperedges, because they provide more constraints for subsequent matches. By considering both the number of unmapped adjacent hyperedges and the size of candidate sets, we can effectively prune the search space.

7 Performance Evaluation

In this section, we conduct experiments to evaluate the effectiveness of our algorithm, MaCH. Our evaluation focuses on two main aspects. First, we compare our algorithm's performance against the state-of-the-art algorithm to demonstrate overall effectiveness of our approach. Second, we analyze individual techniques in our algorithm to show their specific contributions to performance improvement.

Table 2: Statistics of Data Hypergraphs. $|V|$ and $|E|$ denote the number of vertices and hyperedges, respectively. $|\Sigma|$ is the size of the vertex label set ('-' indicates no labels), $a^{\max}$ is the maximum arity, and $\overline{a}$ is the average arity.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$	$a^{\max}$	$\overline{a}$
HC	1,290	336	2	81	35.15
NC	1,161	1,049	-	39	6.17
MA	73,851	5,445	704	1,784	24.19
NS	5,556	6,631	-	187	9.70
CH	327	7,818	9	5	2.33
CP	242	12,704	11	5	2.42
SB	294	21,721	2	99	9.90
EU	1,005	24,520	-	40	3.62
HB	1,494	54,933	2	399	22.15
WT	88,860	65,979	11	25	6.86
CB	1,718	83,105	-	25	8.81
TC	172,738	220,971	160	85	3.18
CM	1,034,876	252,706	-	925	3.02
SA	15,211,989	1,103,218	26,333	61,315	23.67
AR	2,268,231	4,242,421	29	9,350	17.17

7.1 Experimental Setup

Baseline. We primarily compare MaCH with the state-of-the-art hypergraph pattern matching algorithm, HGMatch [44] because HGMatch significantly outperforms other methods such as an extension of a subgraph matching algorithm [20]. We also compare MaCH with the approach of transforming a hypergraph into a bipartite graph and solving the subgraph matching problem. For this approach, we use the state-of-the-art subgraph matching algorithm, GuP [2]. We don't include SubHyMa [38] in our experiments, as the available implementation of SubHyMa[2]is designed to terminate after finding only one embedding.

Datasets. We conduct experiments on 10 real-world hypergraphs from [3], which were previously used to evaluate HGMatch [44], namely 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) [1, 11, 33, 43]. We also evaluate on five unlabeled real-world hypergraphs, namely NDC-classes (NC), NDC-substances (NS), email-Eu (EU), congress-bills (CB), and coauth-MAG-history (CM) [4], by assigning the same label to all vertices. We remove all repeated hyperedges and self-loops from hypergraphs. Additionally, for the MA and SA datasets, which contain multiple labels for vertices, we retain only the first label. Table 2 shows the statistics of the processed datasets.

Queries. We generated query hypergraphs with varying sizes for each dataset, specifically containing 3, 6, 9, 12, or 15 hyperedges. For each dataset and each query size setting, we create a set of 30 query hypergraphs by using a random walk-based approach. The process begins by selecting an initial hyperedge uniformly at random from the data hypergraph. Subsequently, we perform a random walk, uniformly selecting the next hyperedge from the set of

[2]https://github.com/Su-Yuhang/SubHyMa

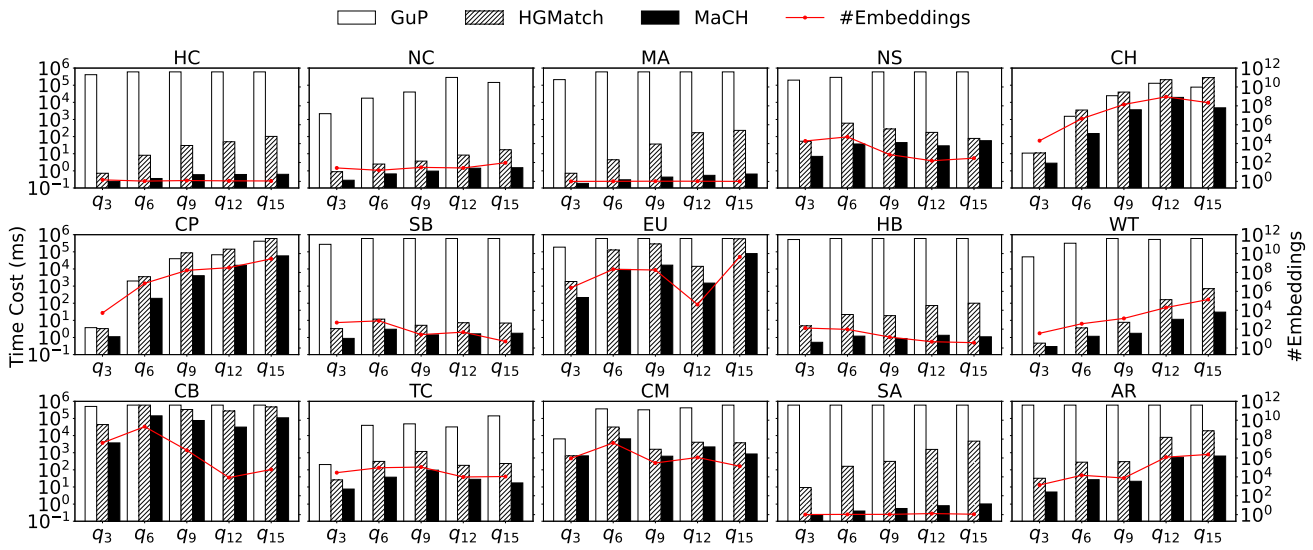


Figure 6: Average query processing time of GuP, HGMatch, and MaCH on different datasets and query sizes. The x-axis represents different query sizes, the left y-axis shows query processing time in milliseconds on a logarithmic scale, and the line graph represents the average number of embeddings (right y-axis) for each query size.

hyperedges adjacent to the current hyperedge. This process continues until the desired number of hyperedges is reached, ensuring that a generated query hypergraph is connected and there is at least one embedding in the data hypergraph. We do not impose any constraints on the number of vertices in the generated query hypergraphs, whereas HGMatch limits the number of vertices to at most 35 in their experiments [44].

Both HGMatch and GuP sometimes incorrectly report zero embeddings, despite the existence of embeddings in the data hypergraph. To maintain a fair comparison, we exclude these queries from our reports on the number of solved queries, query processing time, and memory usage. As a result, the total number of queries per dataset and query size setting may be less than 30 in our experiments due to these exclusions.

Environment. MaCH is implemented in C++. The source code of HGMatch, which is implemented in Rust, was obtained from the authors. GuP is also implemented in Rust and is publicly available. Experiments are conducted on a machine with two Intel Xeon Silver 4114 2.20GHz CPUs and 256GB memory.

In our analysis, we use the geometric mean when referring to average query processing times and other performance metrics. The geometric mean is particularly suitable for our experiments as these metrics vary widely across queries. We set a time limit of 10 minutes for each query and exclude queries that none of the algorithms (GuP, HGMatch, and MaCH) solve within this time limit. When an algorithm fails to solve a query that another algorithm solves, we record 10 minutes for the failing algorithm when calculating the average query processing time.

Preprocessing. Following HGMatch's methodology, we implemented a preprocessing step that builds an index of the data hypergraph prior to query processing. This index consists of signatures of all hyperedges in the data hypergraph and partitions of hyperedges based on their signatures. Unlike HGMatch, we do not build an inverted hyperedge index. This preprocessing is performed using only the data hypergraph, before any query hypergraphs are given.

HGMatch's preprocessing time ranges from 0.9 ms for NC to 9,693.8 ms for AR, while our preprocessing time ranges from 1.2 ms for NC to 6,967.7 ms for AR, generally increasing with the number of hyperedges in the dataset. This preprocessing time is negligible compared to the total query processing time. For example, on AR dataset, preprocessing takes about 7 seconds while processing 30 queries of size 15 takes more than 1.6 hours. For GuP, the preprocessing time for transforming a data hypergraph into a bipartite graph ranges from 0.3 ms for NC to 18,633.4 ms for AR.

The preprocessing time is not included in the query processing times reported in our results, as preprocessing is performed once for each dataset, rather than for each query.

7.2 Comparison with State-of-the-Art Method

Query Processing Time. Figure 6 presents a comparison of the average query processing time between GuP, HGMatch, and MaCH. The time consumption is measured in milliseconds (ms) and displayed on a logarithmic scale. Additionally, a line graph represents the average number of embeddings for each query size. For the CP dataset with query size 15, HGMatch fails to solve any queries within the time limit. Thus, HGMatch's average query processing time is recorded as the time limit of 10 minutes. Likewise, in many cases including SA and AR datasets, GuP fails to solve any queries within the time limit and its average query processing times are recorded as the time limit of 10 minutes.

The query processing time of MaCH increases as the number of embeddings grows, demonstrating a clear correlation with the number of embeddings. In contrast, GuP and HGMatch do not

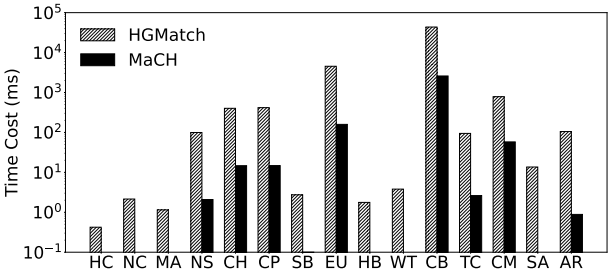


Figure 7: Average verification time of HGMatch and MaCH on different datasets. The y-axis represents the time in milliseconds. Bars with times less than 0.1 ms are omitted.

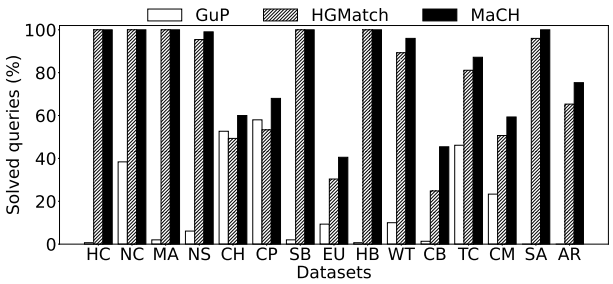


Figure 8: Ratio of solved queries within time limit of 10 minutes on different datasets.

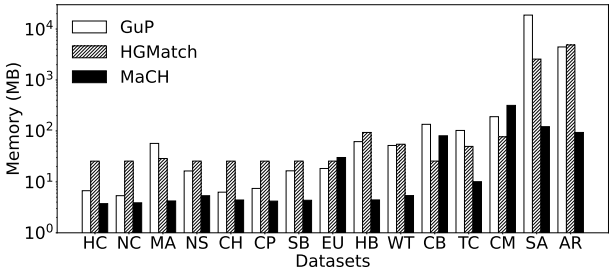


Figure 9: Peak memory consumption in megabytes, averaged over the queries for each dataset.

always follow this trend: for HC, MA, HB, and SA, they show high query processing times even when the number of embeddings is relatively low.

MaCH consistently outperforms its competitors in terms of query processing time, with the performance gap widening for larger query sizes. Notably, MaCH demonstrates particularly effective results for datasets with a large vertex label set and high average arity, i.e., HC, MA, and SA. These properties lead to more distinct hyperedge signatures, resulting in smaller candidate hyperedge sets and fewer embeddings. The performance gap is most evident in the SA dataset for queries of size 15, where MaCH processes queries more than 4,000 times faster than HGMatch and 500,000 times faster than GuP.

The intersection constraint is a necessary and sufficient condition for valid embeddings. Since HGMatch also proposed a necessary and sufficient condition, equivalence of vertex profiles, we compare the two conditions for their efficiency. Figure 7 shows the candidate verification time spent for checking the intersection constraint in MaCH and that for checking the equivalence of vertex profiles in HGMatch (this result includes only queries finished by both algorithms within the time limit).

For datasets whose average verification time of MaCH is more than 0.1 ms (i.e., NS, CH, CP, EU, CB, TC, CM, and AR), HGMatch spends 94.1% to 96.9% of its query processing time on the equivalence of vertex profiles, and MaCH spends 69.4% to 88.0% on the intersection constraint. These datasets require a significant amount of time to solve, highlighting the importance of an effective condition for valid embeddings. As shown in Figure 7, our intersection constraint consistently requires less time across all datasets compared to HGMatch's condition. Particularly, for the AR dataset, checking the equivalence of vertex profiles in HGMatch takes more than 100 times longer than checking the intersection constraint in MaCH.

For remaining datasets, the number of candidates in the CHS after filtering is small, making the verification time of MaCH negligible. As can be seen in Figure 6, these datasets are the least time-consuming for MaCH. We further analyze the effectiveness of our techniques in detail in Section 7.3.

The Ratio of Solved Queries. Figure 8 presents the ratio of solved queries within the time limit of 10 minutes to the total number of queries. This ratio is shown for GuP, HGMatch, and MaCH across all datasets. MaCH successfully solves all the queries that HGMatch or GuP solve, except a few queries in the CM dataset, and outperforms them by solving additional queries in many cases.

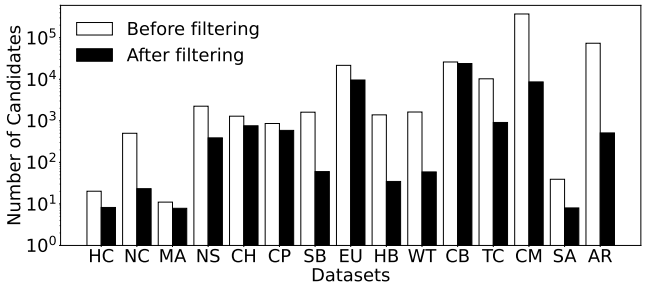
For HC, MA, and SA, which typically have a low number of embeddings per query, MaCH and HGMatch generally complete the search within the time limit. GuP, on the other hand, solves almost no queries on these datasets. GuP tends to perform worse on datasets having high average arity (HC, MA, HB, SA, and AR) than those having low average arity (NC, CH, CP, EU, WT, TC, and CM).

Although EU, CB, and CM datasets are not the largest, they are particularly challenging due to the absence of vertex labels. This absence significantly reduces the number of distinct hyperedge signature, increasing the number of embeddings per query. CH and CP datasets also present significant challenges for MaCH and HGMatch. They also have the high number of embeddings per query, caused by low maximum arity and limited number of labels. While GuP shows better performance than HGMatch on CH and CP, MaCH still outperforms GuP on these datasets.

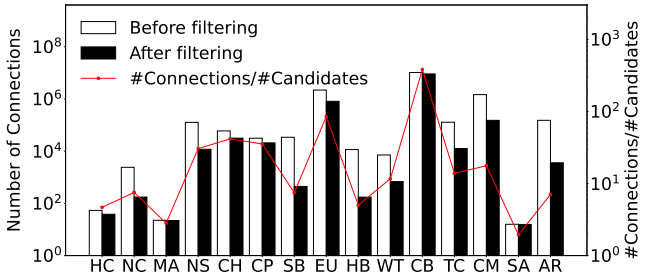
Memory Consumption. Figure 9 illustrates the peak memory consumption of GuP, HGMatch, and MaCH averaged over the queries that are solved by at least one algorithm. The memory consumption is measured in megabytes. In general, MaCH exhibits comparable or better memory efficiency than GuP and HGMatch. While memory consumption tends to increase with the size of the data hypergraph for all algorithms, MaCH demonstrates significantly lower memory consumption on the two largest datasets (SA and AR).

7.3 Effectiveness of Individual Techniques

Filtering by Connectivity Constraint. Figure 10 illustrates the effect of our filtering technique on the candidate hyperedge space.



(a) Number of candidates in the candidate hyperedge space.



(b) Number of connections in the candidate hyperedge space. The left y-axis shows the number of connections. A line graph represents the average number of connections per candidate (right y-axis) after filtering.

Figure 10: Average size of the candidate hyperedge space before and after filtering on different datasets

The figure compares the average number of candidates and connections in the CHS before and after applying our filtering method for queries. A line graph in Figure 10b represents the average number of connections per candidate after filtering.

In all datasets, our filtering technique reduces the number of candidates and the number of connections significantly. The effect of our filtering is especially evident in the AR dataset. In AR, filtering reduces the number of candidates by 99.3% and the number of connections by 97.7%, eliminating a significant portion of candidates and connections from the initial CHS. It is due to the relatively large average arity and large number of labels in the dataset. These characteristics result in diverse signatures for the intersection of two hyperedges, which means that more candidates are likely to be filtered out by the connectivity constraint.

For CH and CP datasets, which have the small average arity and small number of labels (opposite to AR), our filtering still demonstrates its effectiveness. It reduces candidates by 41.1% and 32.1% respectively, and connections by 47.2% and 34.1%.

For the CB dataset, which has the highest number of connections per candidate after filtering, the filtering process is less significant, highlighting the importance of efficient cleaning during the matching process.

Cleaning Techniques. Figure 11 demonstrates the effect of our match-and-clean framework on the average query processing time for each dataset. The first bar shows the query processing time without any cleaning techniques. In this algorithm, candidates are

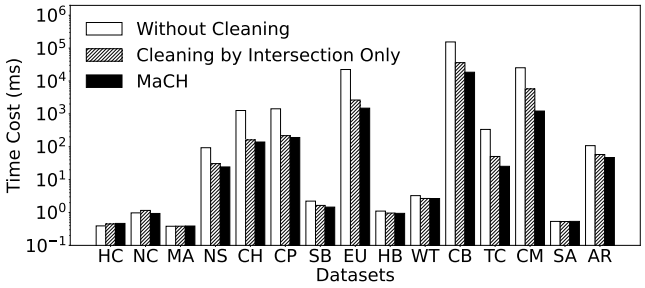


Figure 11: Average query processing time with and without cleaning techniques. The y-axis shows time in milliseconds. The first bar shows time without any cleaning techniques, the second bar shows time when only cleaning by intersection constraint is applied, and the third bar shows time when both intersection and connectivity constraints are applied.

not guaranteed to be M-compatible, meaning some candidates may not be valid for extending the partial embedding. Thus, before extending the partial embedding, we check the intersection constraint for each candidate. The second bar represents the query processing time when only cleaning by the intersection constraint is applied, without cleaning by the connectivity constraint. The third bar shows the query processing time when both cleaning techniques are applied. In general, adding the connectivity constraint is more advantageous because it is fast ($O(1)$ time per candidate) and it eliminates a considerable number of candidates in the CHS.

The results demonstrate that the match-and-clean framework is particularly effective in datasets with higher time consumption, i.e., large unlabeled datasets (EU, CB, and CM). For instance, in the CM dataset, our algorithm with both cleaning techniques is 20.7 times faster than without cleaning and 4.7 times faster than with cleaning by the intersection constraint only.

The CH, CP, and CB datasets, where more than half of the candidates remain after filtering, show notable performance improvements through the cleaning process. In these datasets, as a partial embedding is extended, our cleaning techniques capture complex connectivity patterns involving three or more hyperedges that the filtering cannot capture. Consequently, the cleaning process during matching substantially improves performances for these datasets.

For HC, NC, MA, and SA, the impact of these cleaning techniques is less significant because the number of candidates in the CHS after filtering is already small.

8 Conclusion

We have proposed a novel hypergraph pattern matching algorithm that utilizes the intersection constraint, candidate hyperedge space, and Match-and-Clean framework. The intersection constraint, as a necessary and sufficient condition for valid embeddings, has potential applications beyond hypergraph pattern matching, such as hypergraph isomorphism [18]. In addition, the combination of CHS and Match-and-Clean is a promising framework for other problems that require effective search space pruning, including subhypergraph random sampling [12] and subhypergraph cardinality estimation [29].

References

[1] Ilya Amburg, Nate Veldt, and Austin R. Benson. 2020. Clustering in graphs and hypergraphs with categorical edge labels. In *Proceedings of The ACM Web Conference*. 706–717.

[2] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[3] Austin R. Benson. 2024. Hypergraph Datasets. https://www.cs.cornell.edu/~arb/data/. Last accessed: 2024-06-05.

[4] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (2018), E11221–E11230.

[5] Christian Bessière. 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* 65, 1 (1994), 179–190.

[6] Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, and Yuanlin Zhang. 2005. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence* 165, 2 (2005), 165–185.

[7] Christian Bessiére, Eugene C. Freuder, and Jean-Charles Regin. 1999. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence* 107, 1 (1999), 125–148.

[8] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the ACM SIGMOD 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 ACM SIGMOD International Conference on Management of Data*. 1199–1214.

[10] Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (1977), 762–772.

[11] Philip S Chodrow, Nate Veldt, and Austin R Benson. 2021. Generative hypergraph clustering: from blockmodels to modularity. *Science Advances* 7, 28 (2021).

[12] Minyoung Choe, Jaemin Yoo, Geon Lee, Woonsung Baek, U Kang, and Kijung Shin. 2024. Representative and Back-In-Time Sampling from Real-world Hypergraphs. *ACM Transactions on Knowledge Discovery from Data* 18, 6 (2024).

[13] Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. 2023. BICE: Exploring Compact Search Space by Using Bipartite Matching and Cell-Wide Verification. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 2186–2198.

[14] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.

[15] Guilherme Ferraz de Arruda, Giovanni Petri, and Yamir Moreno. 2020. Social contagion models on hypergraphs. *Physical Review Research* 2 (2020), 023032. Issue 2.

[16] Moshe Dubiner, Zvi Galil, and Edith Magen. 1994. Faster tree pattern matching. *J. ACM* 41, 2 (1994), 205–213.

[17] 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.

[18] Yifan Feng, Jiashu Han, Shihui Ying, and Yue Gao. 2024. Hypergraph Isomorphism Computation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 5 (2024), 3880–3896.

[19] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

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

[21] 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 ACM SIGMOD International Conference on Management of Data*. 1429–1446.

[22] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 337–348.

[23] Christoph M. Hoffmann and Michael J. O'Donnell. 1982. Pattern Matching in Trees. *J. ACM* 29, 1 (1982), 68–95.

[24] Zite Jiang, Shuai Zhang, Xingzhong Hou, Mengting Yuan, and Haihang You. 2024. IVE: Accelerating Enumeration-Based Subgraph Matching via Exploring Isolated Vertices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 4208–4221.

[25] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2022. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB Journal* 32, 2 (2022), 343–368.

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

[27] Florian Klimm, Charlotte M Deane, and Gesine Reinert. 2021. Hypergraphs for predicting essential genes using multiprotein complex data. *Journal of Complex Networks* 9, 2 (2021), 1–16.

[28] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350.

[29] Quintino Francesco Lotito, Federico Musciotto, Federico Battiston, and Alberto Montresor. 2024. Exact and sampling methods for mining higher-order motifs in large hypergraphs. *Computing* 106, 2 (2024), 475–494.

[30] Alan K. Mackworth. 1977. Consistency in networks of relations. *Artificial Intelligence* 8, 1 (1977), 99–118.

[31] Amine Mhedhbi and Semih Salihoglu. 2018. Optimizing subgraph queries by combining binary and worstcase optimal joins. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 1692–1704.

[32] Roger Mohr and Thomas C. Henderson. 1986. Arc and path consistency revisited. *Artificial Intelligence* 28, 2 (1986), 225–233.

[33] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 188–197.

[34] E. Ramadan, A. Tarafdar, and A. Pothen. 2004. A hypergraph model for the yeast protein complex network. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 189.

[35] Stuart Russell and Peter Norvig. 2021. *Artificial Intelligence: A Modern Approach* (4 ed.). Pearson.

[36] Daniel Sabin and Eugene C. Freuder. 1994. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of International Conference on Principles and Practice of Constraint Programming (CP)*. 10–20.

[37] L. Siddharth, Lucienne Blessing, and Jianxi Luo. 2022. Natural language processing in-and-for design research. *Design Science* 8 (2022), e21.

[38] 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.

[39] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[40] Shixuan Sun and Qiong Luo. 2022. Subgraph Matching With Effective Matching Order and Indexing. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 491–505.

[41] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: a holistic approach to subgraph query processing. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 176–188.

[42] Shulong Tan, Ziyu Guan, Deng Cai, Xuzhen Qin, Jiajun Bu, and Chun Chen. 2014. Mapping users across networks by manifold alignment on hypergraph. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 159–165.

[43] Nate Veldt, Austin R. Benson, and Jon Kleinberg. 2020. Minimizing Localized Ratio Cut Objectives in Hypergraphs. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 1708–1718.

[44] 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 IEEE International Conference on Data Engineering (ICDE)*. 2063–2076.

[45] 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.

[46] Lingling Zhang, Zhiwei Zhang, Guoren Wang, Ye Yuan, Shuai Zhao, and Jianliang Xu. 2023. HyperISO: Efficiently Searching Subgraph Containment in Hypergraphs. *IEEE Transactions on Knowledge and Data Engineering* 35, 8 (2023), 8112–8125.

[47] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1–29.

A Appendix

A.1 Time Complexity of the Filtering Algorithm

The time complexity of Algorithm 2 is $O(|E_q|^2|E_H|^2)$ as follows. There are $O(|E_q|^2)$ pairs of adjacent hyperedges in the query hypergraph, each initially pushed to the queue during initialization. Additionally, a pair of adjacent hyperedges (e, e') is pushed to the queue when candidates are removed from $C(e')$. Since the number of candidates in $C(e')$ can be up to $|E_H|$, (e, e') is pushed to the queue $O(|E_H|)$ times. When (e, e') is popped from the queue, it checks whether $C(e' \mid e, f)$ is empty for every $f \in C(e)$, which takes $O(|E_H|)$ time in total.

Therefore, the total time complexity is $O(|E_q|^2|E_H|^2)$. In practice, the algorithm's performance is significantly better than this worst-case analysis suggests. This is due to two key factors. First, the number of candidates for each query hyperedge is typically much smaller than $|E_H|$. Second, each pair of hyperedges is rarely pushed to the queue $|E_H|$ times, as candidates are often removed in groups rather than individually.

A.2 Proof of Theorem 6.2

Theorem 6.2. *Given a partial query q' of q and hyperedge mapping $M : E_{q'} \to E_H$, M is a partial embedding if and only if the signature of a cell S matches the signature of $M(S)$ for every subset of query hyperedges $S \in E_{q'}$.*[1]

Proof. ($\Leftarrow$) For any label l, the number of vertices with label l in the intersection $\bigcap_{e \in S} e$ is equal to the sum of the numbers of vertices with label l across all cells that compose this intersection. Given that the signature of a cell S matches the signature of $M(S)$ for every query hyperedge set S, it follows that the signatures of intersections, $Sig(\bigcap_{e \in S} e)$ and $Sig(\bigcap_{e \in S} M(e))$, also match for every S. By Theorem 4.5, M is a partial embedding.

($\Rightarrow$) Conversely, if M is a partial embedding of q', then by Theorem 4.5, $Sig(\bigcap_{e \in S} e) = Sig(\bigcap_{e \in S} M(e))$ for every $S \subseteq E_{q'}$. We can derive that the signature of a cell S matches the signature of $M(S)$ for every query hyperedge set S using the inclusion-exclusion principle. □

A.3 Proof of Theorem 6.6

Theorem 6.6. *Let $M : E_{q'} \to E_H$ be a partial embedding for a partial query q' of q. A candidate $f \in C(e)$ for a query hyperedge $e \in E_q \setminus E_{q'}$ is M-compatible if $\mathcal{I}_q^{M'}(b, l) = \mathcal{I}_H^{M'}(b, l)$ for the bitmap b of every cell S that is a subset of e and every label l.*[1]

Proof. When we extend M to $M' = M \cup \{(e, f)\}$ by mapping e to f, each cell S of $E_{q'}$ that contains vertices in e is divided into two cells as follows.

- $A = S \cap e$, i.e., the cell in S which is a subset of e.
- $B = S \setminus A$, i.e., the cell in S which is not a subset of e.

Since M is a partial embedding, $\mathcal{I}_q^M(b_S, l) = \mathcal{I}_H^M(b_S, l)$ holds for the bitmap b_S of S. Also, $\mathcal{I}_q^{M'}(b_A, l) = \mathcal{I}_H^{M'}(b_A, l)$ for the bitmap b_A of the cell A by the given condition. Since $B = S \setminus A$, we have $\mathcal{I}_q^{M'}(b_B, l) = \mathcal{I}_H^{M'}(b_B, l)$ for the bitmap b_B of the cell B. Therefore, $\mathcal{I}_q^{M'} = \mathcal{I}_H^{M'}$ holds for every cell and label. □