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

Siwoo Song[†], Wonseok Shin[‡], Kunsoo Park*[§], Giuseppe F. Italiano[¶], Zhengyi Yang[◊], Wenjie Zhang[◊]
[†]*Samsung Research, Korea* [‡]*Standigm Inc, Korea* [§]*Seoul National University, Korea*
[¶]*LUISS University, Italy* [◊]*University of New South Wales, Australia*
[†]s6uos.song@samsung.com [‡]wonseok.shin@standigm.com [§]kpark@theory.snu.ac.kr
[¶]gitaliano@luiss.it [◊]{zhengyi.yang,wenjie.zhang}@unsw.edu.au

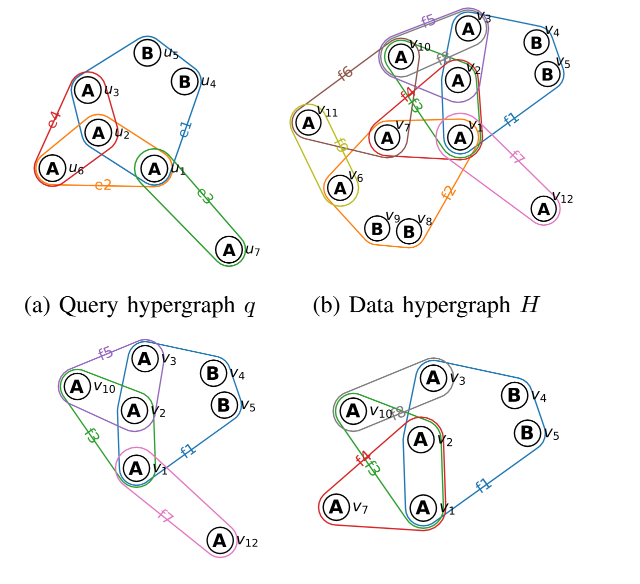
***Abstract*—A hypergraph is a generalization of a graph, in which a hyperedge can connect multiple vertices, modeling complex relationships involving multiple vertices simultaneously. Hypergraph pattern matching, which is to find all isomorphic embeddings of a query hypergraph in a data hypergraph, 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-Filter framework, which interleaves matching and filtering 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 algorithms, by up to orders of magnitude in terms of query processing time.**

***Index Terms*—Hypergraph Pattern Matching, Intersection Constraint, Candidate Hyperedge Space, Match-and-Filter.**

I. 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 [1]–[4]. 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 complex relationships that arise in diverse domains, including chemical reaction networks [5], protein interactions [1], [2], knowledge bases [6], collaboration networks [7], social networks [3], and electronic circuits [8], [9].

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], [11]; if trees, tree pattern matching [12], [13]; if graphs, subgraph matching [14], [15]; if hypergraphs, hypergraph pattern matching (also known as subhypergraph matching [16]). 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).



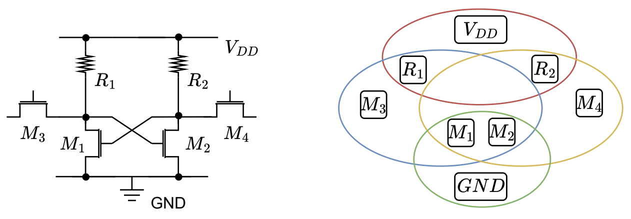
(a) Query hypergraph q (b) Data hypergraph H

(c) Embedding of q in H (d) Wrong mapping

Fig. 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.

Hypergraph pattern matching is essential in various applications. For example, electronic circuits are naturally represented as hypergraphs (Figure 2), where vertices model devices (such as transistors) and a hyperedge models a net connecting multiple devices simultaneously. By hypergraph pattern matching on electronic circuits, we can verify whether patented subcircuits appear in a larger circuit or locate problematic subcircuit patterns that may cause failures. However, hypergraph pattern matching is an NP-hard problem [17], requiring substantial computational time for large hypergraph data or queries.

Existing Approaches and Limitations. There are several approaches for hypergraph pattern matching. One 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



(a) Transistor-level Circuit (b) Hypergraph

Fig. 2: Hypergraph representation of a transistor-level circuit, where vertices are devices (transistors M_1 to M_4, resistors R_1 and R_2, V_{DD}, and GND) and a hyperedges is a net connecting multiple devices simultaneously.

incidence relationships. Although this approach allows us to apply existing subgraph matching algorithms, it fails to utilize the properties specific to hypergraphs, resulting in limited performances [18].

To address such issues, recent works [18]–[20] extend an algorithm for subgraph matching to hypergraph pattern matching instead of transforming the hypergraph. Ha et al. [19] extend the subgraph matching algorithm Turbo$_{\text{ISO}}$ [21], and Yang et al. [16] extend more recent algorithms, namely CFL [15], DAF [14], and CECI [22], to hypergraph pattern matching.

HGMatch [16] 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. However, its methods for generating hyperedge candidates on-the-fly during the matching phase and verifying these candidates through repeated checks of hyperedges consume significant processing time.

Very recently, Qi et al. [23] suggested a verification technique based on overlap intersection graph to solve these inefficiencies. However, their approach still requires set intersection operations, which limits overall performance. Furthermore, its high memory overhead makes it not viable for handling large-scale hypergraphs.

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-Filter for Hypergraph Pattern Matching).

1) We introduce two novel constraints, the *connectivity constraint* for the relationship between two hyperedges (e.g., two query hyperedges e_1 and e_2 share two vertices both labeled 'A') and the *intersection constraint* for the relationships involving three or more hyperedges (e.g., e_1, e_2, and e_3 share a vertex labeled 'A'). 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 1 and Theorem 3). 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.
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-Filter*, which interleaves matching and filtering 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 filtering 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.
   We use both the intersection constraint and the connectivity constraint to filter out invalid candidates in the CHS as the matching progresses. This match-and-filter significantly prunes the search space, leading to a speedup of up to 20 times in query processing time, when compared to the matching without filtering.

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

The source code is available online.[1]

[1]https://gitfront.io/r/swsong/ciungWKRzt8K/MaCH/

II. Preliminaries

A. 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 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 (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 [16], 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.

TABLE I: 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

B. Related Works

Hypergraph Pattern Matching. Ha et al. [19] extends the subgraph matching algorithm, Turbo$_{\text{ISO}}$ [21], to hypergraph pattern matching, introducing a technique called the incident hyperedge structure filter. Their algorithm generates candidates that pass the 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 [16] 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. OHMiner [23] presents an overlap-centric system for hypergraph pattern mining. It constructs a DAG called an Overlap Intersection Graph (OIG) by computing intersections of hyperedges to group vertices that share incident hyperedges. During matching, OHMiner maintains an OIG for partial embeddings to verify the partial embedding. In practice, however, many hypergraph pattern matching algorithms occasionally produce incorrect results or segmentation faults.[2]

There are also problems similar to hypergraph pattern matching, with a different definition of an embedding. SubHyMa [18] 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 [20] 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.

[2]HGMatch's implementation occasionally returns zero embeddings when valid embeddings exist, although we believe that the algorithm described in their paper is theoretically sound. OHMiner exhibits both incorrect results and segmentation faults, indicating that there are also implementation-level issues. GuP has been reported to produce incorrect results in recent literature [24], which we also confirmed in our experiments.

Algorithm 1: MaCH

```
Input  : Query hypergraph $q$, data hypergraph $H$
Output: All embeddings of $q$ in $H$
1 $C_{ini} \leftarrow$ BuildCandidateHyperedgeSpace($q, H$)
2 Filtering($q, C_{ini}$)
3 $M \leftarrow \emptyset$
4 MatchAndFilter($q, H, C, M$)
```

Numerous subgraph matching algorithms have been proposed [14], [15], [21], [25]–[30]. 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. Apart from backtracking algorithms, join-based methods [31], [32] 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 [33], [34].

Subgraph matching and related problems have been extended to property graphs, where each vertex and edge can have multiple labels and key-value pairs (properties) [35]–[37]. Subgraph matching on property graphs has been of great interest in graph database systems [38] as it can express rich semantics using graph-structured data.

III. Overview

Constraints for Hypergraph Pattern Matching. We introduce two novel constraints, the *connectivity constraint* and the *intersection constraint*. These constraints are utilized for *filtering*, which is to remove candidates from the candidate hyperedge space (CHS) (Section IV).

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 vertices that have the same label, 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 V).

Match-and-Filter 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-filter framework, which integrates filtering operations directly into the matching process. This framework interleaves matching and filtering operations, utilizing the current partial embedding to filter out invalid candidates from 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 filtering operations directly into the matching process, our framework efficiently handles the complex connectivity patterns in hypergraphs (Section VI).

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.

IV. Constraints for Hypergraph Pattern Matching

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

A. Hyperedge Signature Constraint

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

Definition 3. *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 [16], 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. For simplicity of notation, we will use set notation, e.g. $\{A, A, B\}$, to represent a signature, although it is a multiset. Using this definition, we can now state the Hyperedge Signature Constraint.

Lemma 1 (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.*

B. 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 2 (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 1. *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.*

C. 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 1 (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 $S \subseteq E_q$.*

Example 2. *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.

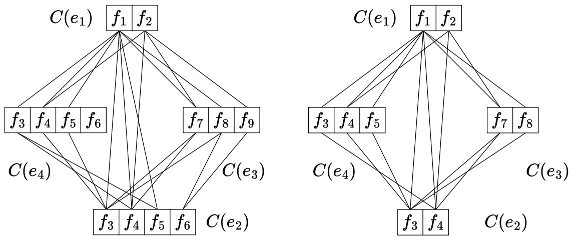
Definition 4. *Let M be a partial embedding for a partial query q' of q and let $e \in E_q \backslash E_{q'}$ be an unmapped query hyperedge. A candidate f for e is* M-compatible *if $M \cup \{(e, f)\}$ is a partial embedding for the partial query induced by $E_{q'} \cup \{e\}$.*

V. Candidate Hyperedge Space

A. Building Candidate Hyperedge Space

Definition 5. *Given a query hypergraph q and a data hypergraph H, a* Candidate Hyperedge Space (CHS) *on q and H consists of 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 $Sig(e \cap e') = Sig(f \cap f')$.*



(a) Initial CHS (b) Filtered CHS

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

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 3. *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 $f \in C(e)$.

B. 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

```
Algorithm 2: FILTERING($q, C$)
  Input : Query hypergraph $q$ and CHS $C$
1 Initialize queue $Q$ with all pairs of adjacent query
    hyperedges
2 while $Q$ is not empty do
3     $(e, e') \leftarrow$ Q.pop()
4     $removed \leftarrow false$
5     foreach $f \in C(e)$ do
6         if $C(e' \mid e, f) = \emptyset$ then
7             Remove $f$ from $C(e)$
8             $removed \leftarrow true$
9     if $removed$ then
10        foreach $e''$ adjacent to $e$ do
11            if $e'' \neq e'$ & $(e'', e) \notin Q$ then
12                $Q.push((e'', e))$
```

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.

Example 4. *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.*

Theorem 2. *Algorithm 2 takes $O((\sum_e |C(e)|)^2)$ time.*

While the number of candidates $|C(e)|$ can reach $|E_H|$ in the worst case, it is typically much smaller than $|E_H|$ in practice.

VI. Match-and-Filter

After finishing initial filtering described in the previous section, Algorithm 3 finds subhypergraph embeddings using a dynamic matching order. It extends the partial embedding

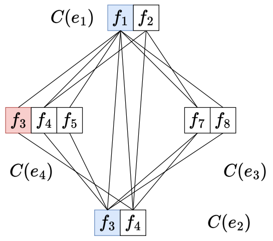
```
Algorithm 3: MatchAndFilter($q, H, C, M$)
  Input: Query hypergraph $q$, data hypergraph $H$, CHS
         $C$, partial embedding $M$
1 if $\exists e \in E_q$ s.t. $C(e) = \emptyset$ then
2     return // No valid embedding
3 $e' \leftarrow$ MatchingOrder($q, C, M$)
4 foreach $f' \in C(e')$ do
5     $M' \leftarrow M \cup \{(e', f')\}$
6     Update $b_q$ and $b_H$
7     if $|M'| = |E_q|$ then
8         Report $M'$
9     else
          // 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 $C(e)$
          // 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        MatchAndFilter($q, H, C, M'$)
18    Restore $b_q$ and $b_H$
19 return
```

M to $M \cup \{(e', f')\}$ by mapping a query hyperedge e' to its candidate hyperedge f'. After each extension, it applies a two-stage filtering process to maintain M-compatibility for all candidates in the candidate hyperedge space.

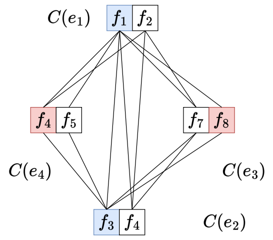
1) Connectivity Constraint Check: A fast, coarse-grained filtering technique that quickly eliminates candidates incompatible with the current partial embedding in $O(1)$ time per candidate (lines 10–13).
2) Intersection Constraint Check: A more thorough, fine-grained filtering, ensuring that all remaining candidates are M-compatible. As we will show in Theorem 5, 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 filtering 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

Fig. 4: Match-and-Filter 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 filtered by connectivity constraint in (a) and intersection constraints in (b).

Algorithm 4: $\mathsf{CheckIntersection}(q, H, C, M, e, f)$

```
  Input: Query $q$, data $H$, CHS $C$, partial embedding
         $M$, query hyperedge $e$, data hyperedge $f$
1 foreach $u \in e$ do
2     $b_q[u] \leftarrow b_q[u] + 2^{pos(e)}$
3 foreach $v \in f$ do
4     $b_H[v] \leftarrow b_H[v] + 2^{pos(e)}$
5 $S_q^+ \leftarrow [(b_q[u], L_q[u]) \mid u \in e]$
6 $S_H^+ \leftarrow [(b_H[v], L_H[v]) \mid v \in f]$
7 if $sort(S_q^+) \neq sort(S_H^+)$ then
8     Remove $f$ from $C(e)$
9 Restore IHBs
```

A. Filtering by Connectivity Constraint

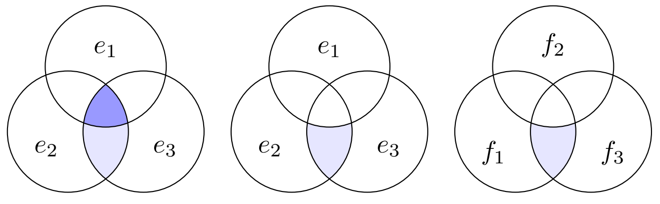
Lines 10–13 of Algorithm 3 are to apply the connectivity constraint to the candidates in CHS after a new hyperedge pair (e', f') is added to the partial embedding M.

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.

Example 5. *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 remove invalid candidates from 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 $f_3 \in C(e_2)$.*

B. Filtering by Intersection Constraint

Algorithm 4 is to apply the intersection constraint to the candidate $f \in C(e)$ for the partial embedding M. It filter out the candidate from CHS if it is not M-compatible. To check the M-compatibility of a candidate, we use Theorem 1, 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.



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

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

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 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 3. *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 $S \in E_{q'}$.*

OHMiner [23] gives an observation which is similar to the above theorem. However, our approach for verification is completely different from OHMiner. While OHMiner computes set intersections repeatedly for vertex sets to verify a candidates, which incurs significant computational overhead, our approach does not utilize set intersections.

Instead, 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 (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. *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 $M : E_{q'} \to E_H$, $\mathcal{I}_q^M(b, l)$ (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 7. *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 3. 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 4. *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.*

Example 8. *Consider Figure 4b with a partial embedding $M = \{(e_1, f_1), (e_2, f_3)\}$. We remove invalid candidates 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 filtering, each of e_3 and e_4 has only one remaining candidate. Mapping these candidates results in the embedding shown in Figure 1c.*

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.

Theorem 5. *The running time of Algorithm 4 is $O(|e|)$.*

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

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.

C. Matching Order

In subgraph matching, two well-known heuristics are the degree heuristic [32], [39] and the candidate size heuristic [14], [27]. The degree heuristic prioritizes query vertices with the highest degree, and the candidate size heuristic prioritizes query vertices with the smallest number of candidates. For subgraph matching on property graphs, van Leeuwen et al. [37] suggest various matching orders, such as utilizing the number of constraints.

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.

TABLE II: 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	$\|V\|$	$\|E\|$	$\|\Sigma\|$	$a^{\max}$	$\overline{a}$
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
CD	1,930,378	2,170,260	-	280	3.43
AR	2,268,231	4,242,421	29	9,350	17.17
AM	13,262,573	4,809,176	258	2,392	7.4

VII. 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 algorithms to demonstrate overall effectiveness of our approach. Second, we analyze individual techniques in our algorithm to show their specific contributions to performance improvement.

A. Experimental Setup

Baseline. We primarily compare MaCH with HGMatch [16] because HGMatch significantly outperforms other methods such as an extension of a subgraph matching algorithm [19]. We also include the very recently proposed OHMiner [23] in our main experiments. In addition, we 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 [30].

Datasets. We conduct experiments on 10 real-world hypergraphs from [40], which were previously used to evaluate HGMatch [16], 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) [41]–[44]. 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) [45], by assigning the same label to all vertices. In addition, we conduct experiments on the large hypergraphs used in OHMiner, coauth-DBLP (CD) and AMiner (AM). 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 II 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. 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 and OHMiner to at most 40 in their experiments [16], [23].

All competitors (HGMatch, GuP, and OHMiner) occasionally produce incorrect results or segmentation faults. 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. OHMiner and GuP are implemented in C++ and Rust, respectively, and both are 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, OHMiner, 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 7260.0 ms for AM, 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. OHMiner's preprocessing time ranges from 15.9 ms for HC to 139,121.0 ms for AM, and it runs out of memory for AR. 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.

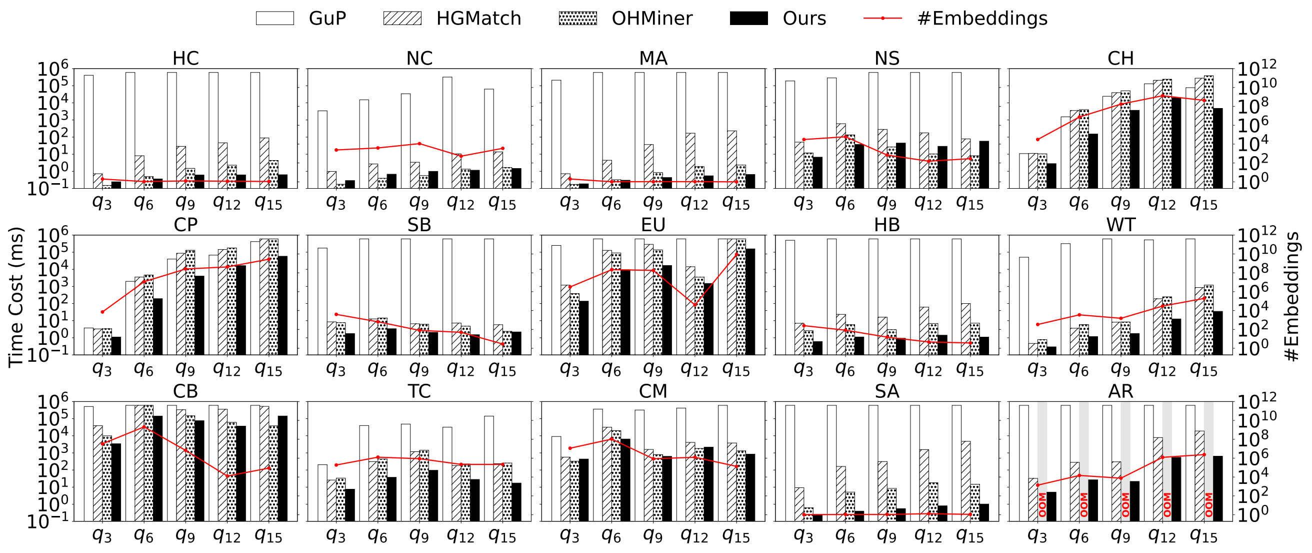


Fig. 6: Average query processing time of GuP, HGMatch, OHMiner, and MaCH on different datasets and query sizes. The x-axis represents different query sizes, the left y-axis shows query processing time in milliseconds on a logarithmic scale, and the line graph represents the average number of embeddings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

B. Comparison with State-of-the-Art Method

Query Processing Time. Figure 6 presents a comparison of the average query processing time between GuP, HGMatch, OHMiner, 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 EU dataset with query size 15 and the CB dataset with query size 6, GuP, HGMatch, and OHMiner solve no queries within the time limit, while MaCH solves 2 and 5 queries, respectively. For these datasets and query sizes, the average query processing times for GuP, HGMatch, and OHMiner are 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. For AR, OHMiner runs out of memory (OOM) and cannot solve any queries.

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 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 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 500,000 times faster than GuP, 4,000 times faster than HGMatch, and 14 times faster than OHMiner. MaCH also shows significant performance advantages for CH with query size 15, especially compared to OHMiner, processing queries more than 16 times faster than GuP, 57 times faster than HGMatch, and 80 times faster than OHMiner.

MaCH sometimes takes more time than OHMiner, i.e., for small queries on HC (q_3) and NC (q_3, q_6, q_9), and for some cases on NS (q_9, q_{12}, q_{15}), CM (q_3, q_{12}), and CB (q_{15}).

As for small queries on HC and NC, these are the least time-consuming cases in our evaluation. In these cases, our initial filtering phase can be an overhead, accounting for 89.8% of query processing time on average. However, as query processing time increases (due to larger query sizes or more challenging datasets), our filtering significantly reduces the number of candidates with a relatively low overhead. On AR, it spends less than 20% of the query processing time, while achieving up to 99% candidate reduction.

Unlabeled hypergraphs such as NS, CM and CB are particularly challenging due to the absence of vertex labels. This absence significantly reduces the number of distinct hyperedge signatures, thus increasing the number of candidates per query. For these unlabeled hypergraphs, MaCH may run slower than OHMiner on some queries. However, Figure 7 shows that MaCH successfully solves more queries within the time limit than OHMiner in all of these challenging datasets. In our experiments, timeout queries were recorded as 10 minutes, which gives an advantage to algorithms that fail to finish with the time limit (10 min). For instance, on NS with query size 15, MaCH successfully solves all queries that OHMiner solves, plus two additional queries for which OHMiner fails to finish. When we ran these two queries without time limits, OHMiner takes 1,234s (20.6 mins) and 15,842s (4.4 hrs), while MaCH takes 47s and 386s (6.4 mins), respectively. Thus, actual performances of MaCH over OHMiner are better than

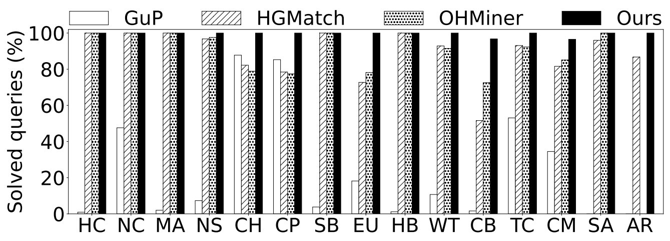


Fig. 7: Ratio of solved queries within time limit of 10 minutes on different datasets.

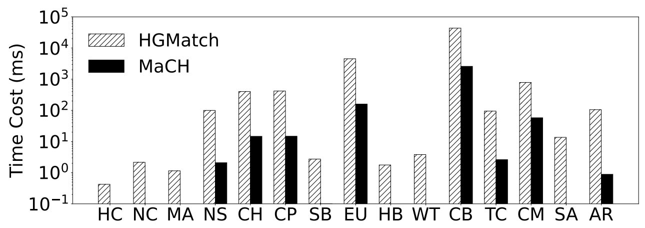


Fig. 8: Average verification time of HGMatch and MaCH on different datasets. The y-axis represents the time in milliseconds. Bars with times less than 0.1 ms are omitted.

those reported in Figure 6.

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 8 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 8, our intersection constraint consistently requires less time across all datasets compared to HGMatch's condition. Particularly, for AR, 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 VII-C.

Ratio of Solved Queries. Figure 7 presents the ratio of queries solved by each algorithm to the queries solved by at least one algorithm within the 10-minute time limit. This ratio is shown for GuP, HGMatch, and MaCH across all datasets. MaCH successfully solves all the queries that HGMatch, OHMiner, or GuP solve, except a few queries in CM and CB datasets, 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, HGMatch, OHMiner, and MaCH 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).

CH and CP datasets also present significant challenges for HGMatch, OHMiner, and MaCH. 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 and OHMiner on CH and CP, MaCH still outperforms GuP on these datasets.

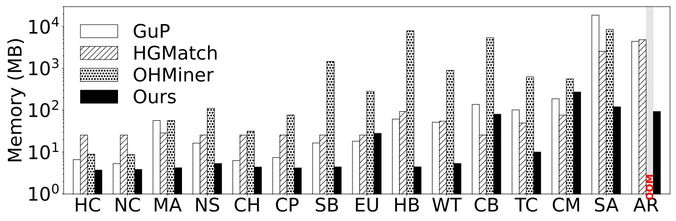


Fig. 9: Peak memory consumption in megabytes, averaged over the queries for each dataset.

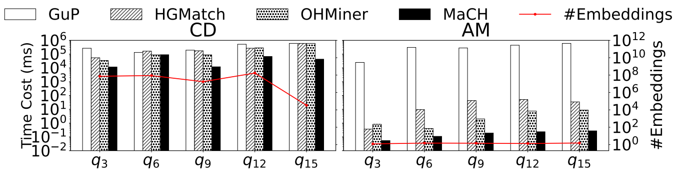


Fig. 10: Average query processing time of GuP, HGMatch, OHMiner, and MaCH on CD and AM.

Memory Consumption. Figure 9 illustrates the peak memory consumption of GuP, HGMatch, OHMiner 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 its competitors. Notably, OHMiner consumes more than 1700 times the memory of MaCH for HB, and runs out of memory (OOM) for AR. 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).

Additional Large Hypergraphs. We conducted additional experiments on the coauth-DBLP (CD) and AMiner (AM) datasets, which are the largest hypergraphs evaluated in the OHMiner paper. Figure 10 presents a comparison of the average query processing time between GuP, HGMatch, OHMiner, and MaCH on the CD and AM datasets. On both datasets, MaCH outperforms its competitors in terms of query processing time. MaCH is particularly effective on AM where the number of vertex labels (258) is large. Specifically, MaCH is more than 1,000,000 times faster than GuP, 100 times faster than HGMatch, and 30 times faster than OHMiner for AM with query size 15. For CD, MaCH is more than 10 times faster than GuP and HGMatch, and more than 7 times faster than OHMiner on query size 9.

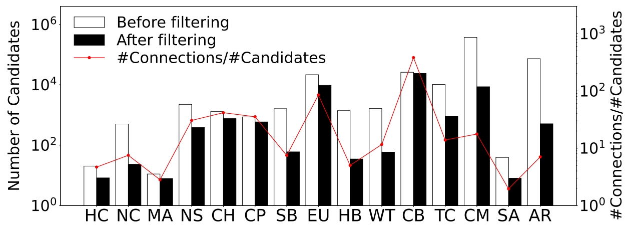


Fig. 11: Average size of the CHS before and after initial filtering on different datasets. The left y-axis shows the number of candidates. A line graph represents the average number of connections per candidate (right y-axis) after initial filtering.

Comparison of Approaches. The approach of transforming a hypergraph into a bipartite graph (GuP) cannot utilize hypergraph-specific properties, leading to poor performance especially on high-arity datasets. HGMatch's on-the-fly candidate generation and repetitive checks in verification cause inefficiency even when the number of embeddings is relatively low. OHMiner generally achieves better performance than GuP and HGMatch but consumes excessive memory, making it not viable for large-scale datasets. In contrast, MaCH addresses these limitations through its efficient constraints, candidate hyperedge space, and Match-and-Filter framework, outperforming all competitors while maintaining comparable or better memory consumption.

C. Effectiveness of Individual Techniques

Initial Filtering by Connectivity Constraint. Figure 11 illustrates the effect of our filtering technique on the candidate hyperedge space. The figure compares the average number of candidates in the CHS before and after applying our filtering method for queries. A line graph 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.

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 filtering during the matching process.

Match-and-Filter Framework. Figure 12 demonstrates the effect of our match-and-filter framework. The first bar shows the query processing time without match-and-filter, where no filtering techniques are used during the matching phase. In this algorithm, candidates may not be guaranteed to be M-compatible. Thus, before extending a partial embedding M with a candidate (e, f) to $M \cup \{(e, f)\}$, we need to verify

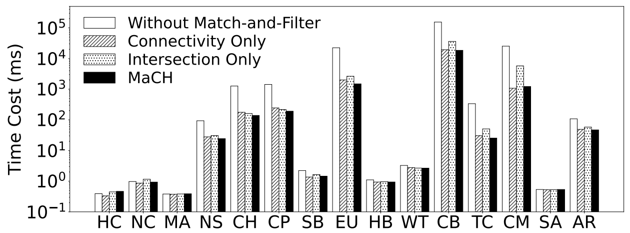


Fig. 12: Average query processing time with and without filtering techniques during matching. The y-axis shows time in milliseconds. The first bar shows time without filtering, the second bar shows time when only filtering by intersection constraint is applied, and the third bar shows time when both intersection and connectivity constraints are applied.

whether (e, f) is M-compatible using the intersection constraint. The second bar shows match-and-filter with filtering only by connectivity constraint is applied. Like the first bar, we still need to verify M-compatibility using the intersection constraint before extending a partial embedding. The third bar shows filtering only by the intersection constraint is applied, thus checking M-compatibility as in MaCH. The fourth bar shows MaCH, where both filtering techniques are applied.

Notably, using either constraint for filtering (second or third bar) demonstrates significant improvement over the baseline (first bar), highlighting the effectiveness of our match-and-filter framework itself. This improvement is particularly shown in datasets with time-consuming datasets, such as EU, CB, and CM. For instance, in the CM dataset, our algorithm with both filtering techniques is 20.7 times faster.

In general, filtering only by connectivity constraint outperforms filtering only by intersection constraint because the connectivity constraint is fast ($O(1)$ time per candidate) and eliminates a considerable number of candidates in the CHS. However, it is important to note that even when filtering only by connectivity constraint is used in match-and-filter, the intersection constraint remains essential for verifying M-compatibility for extending partial embeddings. Using either constraint for filtering (second or third bar) demonstrates significant improvement over the baseline (first bar), highlighting the effectiveness of our match-and-filter framework itself. Overall, the results show that it is beneficial to use both filtering techniques together, especially for challenging datasets such as CH, CP, EU, and CB.

VIII. Conclusion

We have proposed a novel hypergraph pattern matching algorithm that utilizes the intersection constraint, candidate hyperedge space, and Match-and-Filter framework. The intersection constraint, as a necessary and sufficient condition for valid embeddings, has potential applications beyond hypergraph pattern matching, such as hypergraph isomorphism [46]. In addition, the combination of CHS and Match-and-Filter is a promising framework for other problems that require effective search space pruning, including subhypergraph random sampling [47] and subhypergraph cardinality estimation [48].

References

[1] F. Klimm, C. M. Deane, and G. Reinert, “Hypergraphs for predicting essential genes using multiprotein complex data,” *Journal of Complex Networks*, vol. 9, no. 2, pp. 1–16, 2021.

[2] S. Klamt, U.-U. Haus, and F. Theis, “Hypergraphs and cellular networks,” *PLOS Computational Biology*, vol. 5, no. 5, pp. 1–6, 2009.

[3] G. F. de Arruda, G. Petri, and Y. Moreno, “Social contagion models on hypergraphs,” *Physical Review Research*, vol. 2, p. 023032, 2020.

[4] 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*, vol. 22, no. 1, p. 287, 2021.

[5] J. L. Hellerstein, L. P. Smith, L. T. Tatka, S. S. Andrews, M. A. Kochen, and H. M. Sauro, “Discovering subnetworks in sbml models,” *Bioinformatics*, p. btaf482, 09 2025.

[6] B. Fatemi, P. Taslakian, D. Vazquez, and D. Poole, “Knowledge hypergraphs: prediction beyond binary relations,” in *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence (IJCAI)*, 2021, pp. 2191–2197.

[7] R. I. Lung, N. Gaskó, and M. A. Suciu, “A hypergraph model for representing scientific output,” *Scientometrics*, vol. 117, no. 3, pp. 1361–1379, 2018.

[8] B. Li, S. Wang, T. Chen, Q. Sun, and C. Zhuo, “Efficient subgraph matching framework for fast subcircuit identification,” in *Proceedings of the ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2024.

[9] J. Tu, Y. Li, P. Li, P. Xu, Q. Zhang, S. Wan, Y. Sun, B. Yu, and T. Chen, “Smart: Graph learning-boosted subcircuit matching for large-scale analog circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025.

[10] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, “Fast pattern matching in strings,” *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[11] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[12] C. M. Hoffmann and M. J. O’Donnell, “Pattern matching in trees,” *Journal of the ACM*, vol. 29, no. 1, p. 68–95, 1982.

[13] M. Dubiner, Z. Galil, and E. Magen, “Faster tree pattern matching,” *Journal of the ACM*, vol. 41, no. 2, p. 205–213, 1994.

[14] 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 ACM SIGMOD International Conference on Management of Data*, 2019, p. 1429–1446.

[15] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, “Efficient subgraph matching by postponing cartesian products,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016, p. 1199–1214.

[16] Z. Yang, W. Zhang, X. Lin, Y. Zhang, and S. Li, “Hgmatch: A match-by-hyperedge approach for subgraph matching on hypergraphs,” in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2023, pp. 2063–2076.

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

[18] Y. Su, Y. Gu, Z. Wang, Y. Zhang, J. Qin, and G. Yu, “Efficient subhypergraph matching based on hyperedge features,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 6, pp. 5808–5822, 2023.

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

[20] L. Zhang, Z. Zhang, G. Wang, Y. Yuan, S. Zhao, and J. Xu, “Hyperiso: Efficiently searching subgraph containment in hypergraphs,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 8, pp. 8112–8125, 2023.

[21] 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 ACM SIGMOD International Conference on Management of Data*, 2013, p. 337–348.

[22] B. Bhattarai, H. Liu, and H. H. Huang, “Ceci: Compact embedding cluster index for scalable subgraph matching,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2019, p. 1447–1462.

[23] H. Qi, K. Luo, L. He, Y. Zhang, M. Cai, J. Dai, B. He, H. Jin, Z. Zhang, J. Zhao, H. Yue, H. Yu, and X. Liao, “Ohminer: An overlap-centric system for efficient hypergraph pattern mining,” in *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*. New York, NY, USA: Association for Computing Machinery, 2025, p. 621–636.

[24] J. He, Y. Chen, M. Jia, Z. Liu, D. Li, and K.-L. Tan, “Trifmatch: Optimizing large subgraph queries with effective filtering techniques,” 2024. [Online]. Available: http://dx.doi.org/10.2139/ssrn.5030018

[25] L. 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*, vol. 26, no. 10, pp. 1367–1372, 2004.

[26] S. Sun and Q. Luo, “Subgraph Matching With Effective Matching Order and Indexing,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 491–505, 2022.

[27] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, “Fast subgraph query processing and subgraph matching via static and dynamic equivalences,” *The VLDB Journal*, vol. 32, no. 2, p. 343–368, 2022.

[28] Y. Choi, K. Park, and H. Kim, “Bice: Exploring compact search space by using bipartite matching and cell-wide verification,” in *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2023, p. 2186–2198.

[29] Z. Jiang, S. Zhang, X. Hou, M. Yuan, and H. You, “Ive: Accelerating enumeration-based subgraph matching via exploring isolated vertices,” in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2024, pp. 4208–4221.

[30] J. Arai, Y. Fujiwara, and M. Onizuka, “Gup: Fast subgraph matching by guard-based pruning,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2023.

[31] A. Mhedhbi and S. Salihoglu, “Optimizing subgraph queries by combining binary and worstcase optimal joins,” in *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2018, pp. 1692–1704.

[32] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, “Rapidmatch: a holistic approach to subgraph query processing,” in *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2020, p. 176–188.

[33] S. Sun and Q. Luo, “In-Memory Subgraph Matching: An In-depth Study,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1083–1098.

[34] Z. Zhang, Y. Lu, W. Zheng, and X. Lin, “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*, 2024, pp. 1–29.

[35] S. Han and Z. G. Ives, “Implementation strategies for views over property graphs,” *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–26, 2024.

[36] W. Xie, M. Zhang, X. Liao, K. Chen, J. Jiang, and Y. Wu, “Vertexsurge: Variable length graph pattern match on billion-edge graphs,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 361–375.

[37] W. van Leeuwen, G. Fletcher, and N. Yakovets, “A general cardinality estimation framework for subgraph matching in property graphs,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 6, pp. 5485–5505, 2022.

[38] Y. Pang, L. Zou, and M. T. Özsu, “A unified narrative for query processing in graph databases,” in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2025, pp. 4492–4496.

[39] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, “A subgraph isomorphism algorithm and its application to biochemical data,” *BMC Bioinformatics*, vol. 14 Suppl 7, 2013.

[40] A. R. Benson, “Hypergraph datasets,” https://www.cs.cornell.edu/∼arb/data/, 2024, last accessed: 2024-06-05.

[41] N. Veldt, A. R. Benson, and J. Kleinberg, “Minimizing localized ratio cut objectives in hypergraphs,” in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2020, p. 1708–1718.

[42] P. S. Chodrow, N. Veldt, and A. R. Benson, “Generative hypergraph clustering: from blockmodels to modularity,” *Science Advances*, vol. 7, no. 28, 2021.

[43] I. Amburg, N. Veldt, and A. R. Benson, “Clustering in graphs and hypergraphs with categorical edge labels,” in *Proceedings of The ACM Web Conference*, 2020, p. 706–717.

[44] J. Ni, J. Li, and J. McAuley, “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)*, 2019, pp. 188–197.

[45] A. R. Benson, R. Abebe, M. T. Schaub, A. Jadbabaie, and J. Kleinberg, “Simplicial closure and higher-order link prediction,” *Proceedings of the National Academy of Sciences*, vol. 115, no. 48, pp. E11 221–E11 230, 2018.

[46] Y. Feng, J. Han, S. Ying, and Y. Gao, “Hypergraph isomorphism computation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 5, pp. 3880–3896, 2024.

[47] M. Choe, J. Yoo, G. Lee, W. Baek, U. Kang, and K. Shin, “Representative and back-in-time sampling from real-world hypergraphs,” *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 6, 2024.

[48] Q. F. Lotito, F. Musciotto, F. Battiston, and A. Montresor, “Exact and sampling methods for mining higher-order motifs in large hypergraphs,” *Computing*, vol. 106, no. 2, pp. 475–494, 2024.

[49] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu, “Towards understanding the runtime performance of rust,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: Association for Computing Machinery, 2023.

APPENDIX

A. *Proof of Theorem 1*

Theorem 1 (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. $\square$

B. *Proof of Theorem 2*

Theorem 2. *Algorithm 2 takes $O((\sum_e |C(e)|)^2)$ time.*

Proof. A pair of adjacent hyperedges (e, e') is pushed to the queue initially and when candidates are removed from $C(e')$. That is, (e, e') is pushed to the queue $O(|C(e')|)$ 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 $O(|C(e)|)$ time in total. Therefore, the total time complexity is $O(\sum_e \sum_{e'} |C(e)||C(e')|) = O((\sum_e |C(e)|)^2)$. $\square$

C. *Proof of Theorem 3*

Theorem 3. *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 $S \in E_{q'}$.*

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 1, M is a partial embedding.

($\Rightarrow$) Conversely, if M is a partial embedding of q', then by Theorem 1, $Sig(\bigcap_{e\in S} e) = Sig(\bigcap_{e\in S} M(e))$ 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. $\square$

D. *Proof of Theorem 4*

Theorem 4. *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.*

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. $\square$

E. *Memory and Ratio of Solved Queries for additional Large Hypergraphs*

Figure 13 presents a comparison of the ratio of solved queries and the peak memory consumption between GuP, HGMatch, OHMiner, and MaCH on the CD and AM datasets.

The left figure shows the ratio of solved queries within the time limit of 10 minutes to the total number of queries solved by at least one algorithm. MaCH consistently solved more queries compared to its competitors.

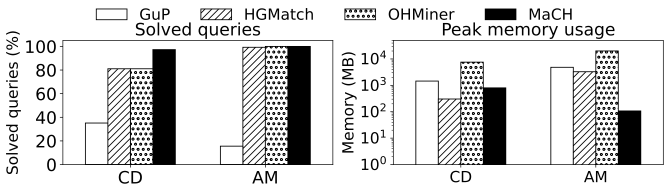


Fig. 13: (Left) Ratio of solved queries within time limit of 10 minutes on CD and AM. (Right) Peak memory consumption in megabytes, averaged over the queries.

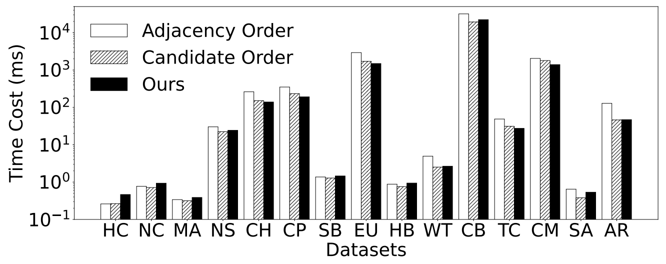


Fig. 14: Average query processing time of Adjacency Order, Candidate Order, and our matching order on different datasets. The y-axis represents the time in milliseconds.

The right figure shows the peak memory consumption averaged over the queries that are solved by at least one algorithm. MaCH exhibits comparable memory efficiency on CD and shows significantly lower memory consumption on AM, where it consumes 32 times less memory than HGMatch and 200 times less memory than OHMiner.

F. *Evaluation of Different Matching Orders*

In subgraph matching, two well-known heuristics are the degree heuristic and the candidate size heuristic. The degree heuristic prioritizes query vertices with the highest degree, and the candidate size heuristic prioritizes query vertices with the smallest number of candidates. In hypergraphs, the degree in graphs corresponds to the number of adjacent hyperedges, and the candidate size corresponds to the number of candidate hyperedges. Thus we consider (1) Adjacency Order, which prioritizes hyperedges with the most unmapped adjacent hyperedges, and (2) Candidate Size Order, which prioritizes hyperedges with the smallest number of candidate hyperedges. Our matching order is a hybrid approach including the criterion that prioritizes query hyperedges with only one candidate.

Figure 14 shows the impact of different matching order strategies on query processing time. The results show that while all three matching orders perform reasonably well, certain datasets benefit from specific strategies. Our hybrid approach achieves good performances on most of the time-consuming datasets (CH, CP, EU, and CM), validating our design choice.

G. *Comparative Analysis with Filtering-First Approaches*

For comparative analysis with traditional filtering-first approaches, we conducted experiments comparing our Match-and-Filter framework against filtering-first strategies.

Traditional filtering-first approaches, such as Ha et al. [19] apply filtering before matching begins. We implemented an enhanced version of adjacency-based filtering of Ha et al., which filters out candidates based on the number of adjacent hyperedges with matching signatures (stronger than their arity-based filter).

Figure 15 shows the comparison: (1) MaCH without match-and-filter framework, (2) MaCH without match-and-filter + adjacency filtering (AdjFilter Without Match-and-Filter), and (3) MaCH. Both (1) and (2) are traditional filtering-first approaches, where all filtering occurs before matching begins.

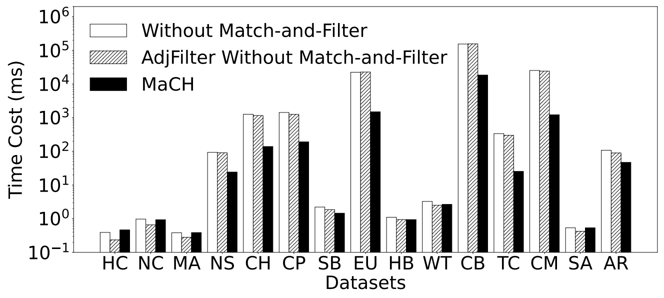


Fig. 15: Average query processing time of traditional filtering-first approaches versus MaCH. Bars show MaCH without Match-and-Filter, Adjacency Filter without Match-and-Filter, and MaCH. The y-axis shows time in milliseconds.

The results show the limitations of filtering-first approaches. While additional adjacency filtering provides modest improvements on small datasets (HC, NC, MA), it fails to improve performances on challenging datasets like EU, CB, and CM.

This occurs because initial filtering cannot capture which candidates will become invalid as the partial embedding grows during matching. Complex connectivity patterns involving multiple hyperedges only become apparent during the matching process. Our Match-and-Filter framework achieves 19.8 times speedup over AdjFilter Without Match-and-Filter on the CM dataset (Figure 15). These results validate that Match-and-Filter better handles the complex connectivity patterns inherent in hypergraphs.

H. *Performance Differences between Rust and C++*

We converted the Rust code of HGMatch (one of main competitors) to C++, and compared it with the original Rust version by authors. We tried to translate the code as faithfully as possible. Figure 16 shows the query processing time for queries on HC, NC, MA, SB, HB, and TC datasets. As shown in the figure, Rust and C++ versions have similar performances, with the C++ version being marginally faster (1.013 times in geometric average across all queries).

Both C++ and Rust are system programming languages with comparable performances. According to Zhang et al. [49], Rust programs run 1.77 times slower than C on average. Similarly, the Computer Language Benchmarks Game[3] reports how many times slower each language is on average relative to the fastest implementation for each benchmark (Figure 17). The result shows that both Rust and C++ are slower than C, but they have similar performances. Therefore, we kept the original Rust code of HGMatch by authors in our experiments.

In addition, the performance differences we observe far exceed what could be attributed to language choice alone. Our algorithm (MaCH) and OHMiner, which shows the second-best performance in our experiments, are both implemented in C++. HGMatch and GuP are implemented in Rust. Our experiments (Figure 6) show that MaCH outperforms HGMatch

[3]https://benchmarksgame-team.pages.debian.net/benchmarksgame/box-plot-summary-charts.html, Last accessed: Sep 1, 2025

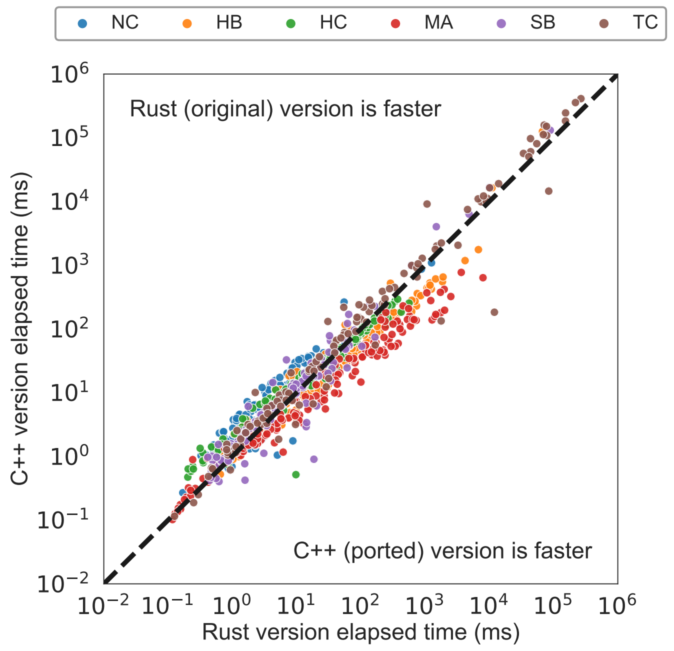


Fig. 16: Comparison of query processing time for queries on HC, NC, MA, SB, HB, and TC datasets. The x-axis and y-axis represent the query processing time of Rust version (HG-Match code obtained from authors) and C++ version (ported), respectively.

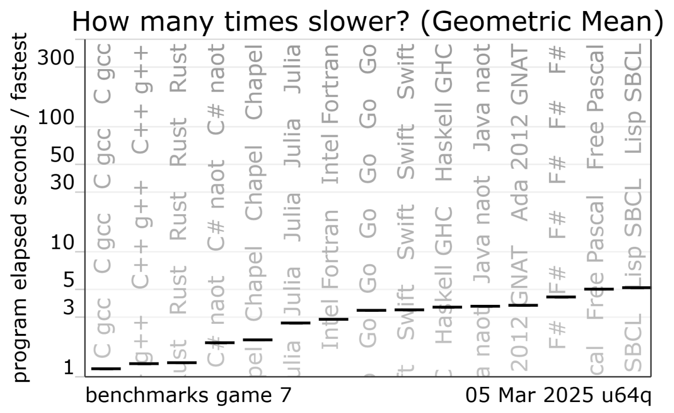


Fig. 17: Running time comparison of programming languages. Numbers show how many times slower each language is on average relative to the fastest implementation for each benchmark (Lower is better). Data from Computer Language Benchmarks Game.

by up to three orders of magnitude (e.g., in the SA dataset for queries of size 15) and it outperforms GuP even more than that.