HESM: A Hyperedge Embedding-Based Subhypergraph Matching Method

Ji Li, Yijie Zhang, Guang Lu, Jiaquan Li, Chuanwen Li(✉)

Northeastern University, Shenyang, China
{2190123,2390236,2010656,2210708}@stu.neu.edu.cn,
lichuanwen@mail.neu.edu.cn

Abstract. Hypergraphs generalize traditional graphs by allowing hyperedges to connect any number of vertices, enabling the representation of complex relationships. Subhypergraph matching queries aim to identify all subhypergraphs within a hypergraph that are isomorphic to a given query hypergraph. However, existing subhypergraph matching methods often fail to leverage the higher-order structure of hypergraphs efficiently. At the same time, these methods typically require traversing each hyperedge in the data hypergraph to find candidate sets for query hyperedges, leading to high computational and time costs. In this paper, we introduce a lossless transformation structure called the hypergraph mapping graph. Leveraging this structure, we propose a hyperedge embedding method that ensures if a hyperedge in the query hypergraph matches one in the data hypergraph, the embedding of the data hyperedge dominates that of the query hyperedge. We then apply spatial indexing to the hyperedge embeddings of the data hypergraph, enabling direct retrieval of query candidate sets without requiring traversal of the data hypergraph. Finally, we present a parallel matching method that capitalizes on GPU parallelism to accelerate the enumeration process. Experimental results demonstrate that our proposed method significantly outperforms existing approaches.

Keywords: Hypergraph · Subhypergraph matching query · Hyperedge embedding · Spatial indexing · GPU.

1 Introduction

Graphs are widely used in various fields such as chemistry, biology, and sociology [1-3] to represent complex relationships between objects. With the advent of big data and advanced data mining technologies, the scale and complexity of data have grown dramatically, making traditional graphs insufficient. Therefore, traditional graphs can no longer meet the increasing demands. The hypergraph is a generalization of the graph. Hypergraphs extend traditional graphs by allowing hyperedges to connect multiple vertices, thus effectively representing complex, higher-order interactions.

The subhypergraph matching problem is to find subhypergraphs on the data hypergraph H that are isomorphic to the query hypergraph q. As illustrated in

Fig. 1, the subhypergraph matching query result of the query hypergraph q on the data hypergraph H is $\{e_{q0} \to e_{H1}, e_{q1} \to e_{H2}, e_{q2} \to e_{H3}\}$ (the red dashed ellipses in Fig. 1 (b)).

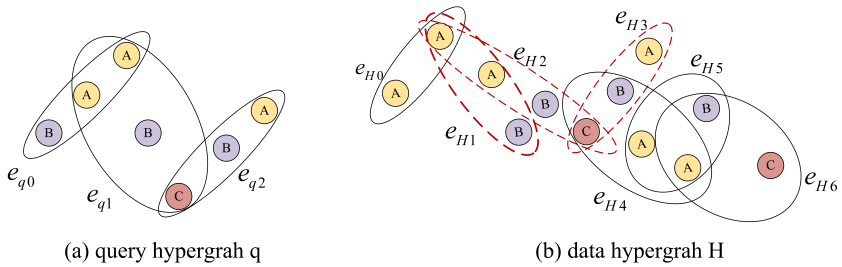

Fig. 1. Example of a subhypergraph matching query.

The subhypergraph matching query has been widely applied in real-world applications. In biological network mining, hypergraph queries enable biologists to understand the interrelationships and interactions between proteins [4–7]. Similarly, hypergraphs can also be applied to gene regulatory networks, disease-related networks, and drug-target interaction networks [8, 9].

Traditional approaches often rely on transforming hypergraphs into bipartite graphs to leverage subgraph matching algorithms. In a bipartite graph, the upper vertices represent hyperedges, the lower vertices represent the hypergraph's vertices, and the edges indicate the inclusion relationships between them. While this preserves hypergraph information, it significantly increases graph size, resulting in high computational overhead.

Currently, there are two methods for subhypergraph matching queries. One method is to extend Ullmann's backtracking algorithm in subgraph matching by modifying the edge constraints into hyperedge constraints [10]. However, this method does not fully utilize the high-order information in the hypergraph, which results in the inability to effectively filter the hypergraph, resulting in an excessively large search space.

Another method is to perform subhypergraph queries based on higher-order information from hypergraphs. Ha [11] and Su [12] filter the data hypergraph using the features of vertices and hyperedges in the hypergraph. Yang [13] utilizes GPU to accelerate subhypergraph matching queries. However, these methods require computing vertex and hyperedge information during matching, which increases the computational cost.

Therefore, in this paper, we introduce a lossless transformation structure for hypergraphs, referred to as the hypergraph-mapping graph. Based on this structure, we propose a hyperedge embedding method. When a hyperedge in the data hypergraph matches a hyperedge in the query hypergraph, we ensure that the embedding of the data hyperedge dominates the embedding of the matching query hyperedge. Subsequently, we use spatial indexing methods to build an index for the hyperedge embeddings of the data hypergraph. When searching for the candidate set of the query hyperedge, we can directly query the

index without having to traverse the entire data hypergraph.Finally, we present a parallel matching method that leverages GPU parallelism to accelerate the enumeration process.

Our main contributions are summarized as follows:

- We propose a structure for mapping hypergraphs into graphs called the *hypergraph-mapping graph*, which maps hypergraphs into graphs losslessly.
- Based on the hypergraph-mapping graph, we propose a hyperedge embedding method. When a hyperedge in the data hypergraph matches a hyperedge in the query hypergraph, we ensure tha the embedding of the data hyperedge dominates the embedding of the matching query hyperedge.
- We employ spatial indexing methods to create indexes for the hyperedge embeddings of the data hypergraph, referred to as the *hyperedge embedding index*. During subhypergraph matching, candidate set query can be quickly and accurately performed directly on the index without the need to traverse the data hypergraph.
- We propose a parallel matching method based on edge connections, that performs parallel concatenation on each branch of the edge spanning tree of the hypergraph-mapping graph. This method fully leverages the parallel computing capabilities of GPUs to improve matching efficiency.
- We conduct extensive experiments on numerous hypergraph datasets, and the experimental results demonstrate that our method is far superior to other state-of-the-art methods.

2 Preliminaries

2.1 Problem Statement

Definition 1 (Graph). *A graph is represented by $G = (V, E, L)$, where V is a set of vertices, E is a set of edges between vertices, and L is a label function that associates each vertex $v \in V$ with a set of labels.*

Definition 2 (Hypergraph). *A hypergraph is represented by $H = (V, E, L)$, where V is a finite set of vertices, E is a set of non-empty subsets of V , and L is a label function that associates each vertex $v \in V$ with a set of labels.*

Definition 3 (Subhypergraph Isomorphism). *Given a query hypergraph q and a data hypergraph H, q is a subhypergraph isomorphic to H if and only if there is an injective mapping $f : V(q) \to V(H)$ such that:*

1. $\forall u \in V(q)$, $L_q(u) = L_H(f(u))$, *where* $f(u) \in V(H)$.
2. $\forall e_q = \{u_1, u_2, \ldots, u_k\} \in E(q)$, $\exists e_H = \{f(u_1), f(u_2), \ldots, f(u_k)\} \in E(H)$.

Problem 1 (Subhypergraph Matching Query). The subhypergraph matching query returns all subhypergraphs that are isomorphic to q on H.

2.2 Subgraph Matching Query Algorithms

Existing subgraph matching algorithms can be broadly categorized as *join-based* or *backtracking search-based*. Join-based algorithms decompose the query graph into smaller substructures, match them independently, and join the results to form the final matches. Backtracking search-based algorithms incrementally explore all possible mappings, reducing intermediate results but incurring high enumeration costs.

The generic subgraph matching [14] includes filtering, ordering, and enumeration stages. The filtering stage generates a complete candidate vertex set $C(u)$ defined in Definition 4. The ordering stage generates a matching order. The enumeration stage recursively enumerates all results.Subhypergraph matching, in contrast to subgraph matching, has been less extensively studied. Existing research on hypergraphs primarily focuses on extracting the features of hyperedges for filtering, with the overall approach following a backtracking search-based framework for subgraph matching[11–13].

Definition 4 (Complete Candidate Vertex Set). *Given q and G, a complete candidate vertex set $C(u)$ of $u \in V(q)$ is a set of data vertices such that for each $v \in V(G)$, if (u, v) exists in a match from q to G, then v belongs to $C(u)$.*

3 Hyperedge Embedding

3.1 Hyperedge Mapping

Definition 5 (Hyperedge Signature). *The signature of a hyperedge e, denoted as $S(e)$, is an ordered multiset of the labels of vertices in e,i.e., $S(e) = multiset\{l(v) : v \in e\}$.*

For example, in Fig. 1, $S(e_{q0}) = [A, A, B]$. We map hyperedges in the hypergraph to vertices using hyperedge signatures.

Definition 6 (Hypergraph-mapping Graph). *In the hypergraph-mapping graph, vertex labels are the hyperedge signatures of the hyperedges in the hypergraph, and the edge labels are the ordered sets of labels of the overlapping vertices of the hyperedges in the hypergraph.*

We use hypergraph-mapping graphs to map hypergraphs to traditional graphs. The vertices in the hypergraph-mapping graph represent hyperedges in the hypergraph, and the vertex labels are the hyperedge signatures of the hyperedges. If two hyperedges in the hypergraph have overlapping vertices, then there is an edge between the corresponding vertices of the hypergraph-mapping graph, and the label of this edge is an ordered set of overlapping vertex labels. By matching the vertex labels of the hypergraph-mapping graph, the integrity of hyperedge semantics can be ensured. By matching the edge labels of the hypergraph-mapping graph, the integrity of hyperedge structures can be ensured.

For example, the hypergraph-mapping graph of the query hypergraph q and data hypergraph H in Fig. 1 is shown in Fig. 2.

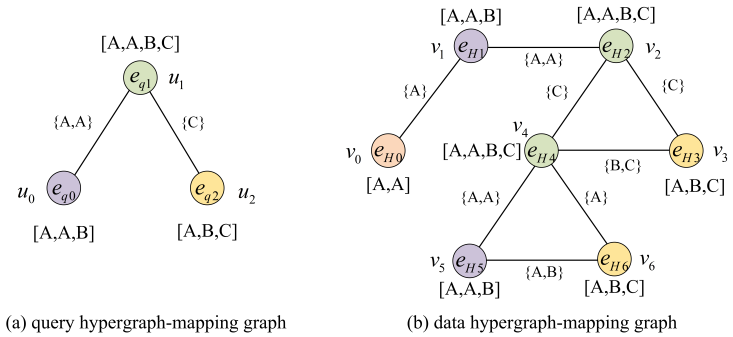

Fig. 2. Example of hypergraph-mapping graph

Definition 7 (Subhypergraph Isomorphism Redefinition). *Given a query hypergraph-mapping graph $q_m = (V_{qm}, E_{qm}, LV_{qm}, LE_{qm})$ and a data hypergraph-mapping graph $H_m = (V_{Hm}, E_{Hm}, LV_{Hm}, LE_{Hm})$, q_m is subhypergraph isomorphic to H_m if and only if there is an injective mapping $f : V(q_m) \to V(H_m)$ such that:*

1. $\forall u \in V_{qm}$, $LV_{qm}(u) = LV_{Hm}(f(u))$, *where* $f(u) \in V_{Hm}$.
2. $\forall (u_i, u_j) \in E_{qm}$, $\exists (f(u_i), f(u_j)) \in E_{Hm}$ *and* $LE_{qm}(u_i, u_j) = LE_{Hm}(f(u_i), f(u_j))$.

Through the hypergraph-mapping graph, we transform the subhypergraph matching query into a traditional subgraph query problem of an undirected graph with labeled vertices and labeled edges. In the hypergraph-mapping graph q_m, the vertex u_i corresponds to the hyperedge e_{qi} in the query hypergraph q, and the vertex v_j in the data hypergraph-mapping graph H_m corresponds to the hyperedge e_{Hj} in the data hypergraph H. In the subsequent sections of this paper, we use u and v to represent hyperedges.

It is worth mentioning that if the vertices in the hypergraph are unlabeled, it is still possible to transform it into a hypergraph-mapping graph. In this case, the vertex labels in the hypergraph-mapping graph represent the number of vertices contained in the hyperedges, and the edge labels represent the number of vertices shared between two overlapping hyperedges.

The space complexity of converting a hypergraph to a bipartite graph is $O(m + n + m \cdot n)$, where m represents the number of hyperedges, n represents the number of vertices. The space complexity of converting a hypergraph to a hypergraph-mapping graph is $O(m + m^2)$. Therefore, hypergraph-mapping graphs are particularly suitable when the number of hyperedges is much smaller than the number of vertices, when hyperedges have minimal overlap, and when the hyperedges are large. At the same time, converting to a bipartite graph causes a greater inflation for query hypergraphs, as the increase in the number of queried vertices significantly boosts the query time. In contrast, the hypergraph-mapping graph does not result in an increase in the number of queried vertices.

3.2 Hyperedge Embedding

Definition 8 (N-layer Hyperedge Path Spanning Tree). *An n-layer hyperedge path spanning tree is a tree composed of paths of length n starting from a vertex v in the hypergraph-mapping graph G_m, denoted as $HPST(G_m, v, n)$. Each path contains no repeated vertices. The vertex labels in $HPST(G_m, v, n)$ represent the hyperedge signatures, and the edge labels represent the ordered sets of labels of the overlapping vertices between two hyperedges.*

Theorem 1. *[16] Given a graph G with n vertices and a graph q with m vertices ($n \geq m$), their adjacency matrices are denoted as M_G and M_q respectively. For matrix M_G, its eigenvalues are $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. For matrix M_q, its eigenvalues are $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_m$. If q is isomorphic to a subgraph of G, then $\beta_i \leq \lambda_i$, $(i = 1, \ldots, m)$.*

Definition 9 (Hyperedge Eigenvalues Embedding). *Given an n-layer hyperedge path-spanning tree $HPST(G_m, v, n)$. Let the eigenvalues of M_{G_m} be $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_s$, where s is the number of vertices in $HPST(G_m, v, n)$. We use t to represent the maximum number of vertices in $HPST(G_m, v, n)$. The hyperedge eigenvalue embedding of the vertex v_j is defined as $HEE_t(G_m, v, n) = [\lambda_1, \lambda_2, \ldots, \lambda_t]$, where each λ_i represents an eigenvalue. If the number of eigenvectors is less than t, then pad with very small values.*

Theorem 2. *Given two hypergraph-mapping graphs q_m and H_m. q_m is subhypergraph isomorphic to H_m under the injective function f. For any vertex u_i in q_m, $HEE_{t_2}(q_m, u_i, n) = [\lambda_1, \lambda_2, \ldots, \lambda_{t_2}]$. There exists the vertex v_j in H_m, where $v_j = f(u_i)$, $HEE_{t_1}(H_m, v_j, m) = [\beta_1, \beta_2, \ldots, \beta_{t_1}]$. If $n \leq m$, it must satisfy that $\lambda_i \leq \beta_i$, for $i = 1, \ldots, t_2$.*

Proof. Since u_i and v_j have a matching relationship, $HPST(q_m, u_i, n)$ is an induced sub-tree of $HPST(H_m, v_j, m)$. Therefore, $HPST(q_m, u_i, n)$ is subgraph isomorphic to $HPST(H_m, v_j, m)$. Based on Theorem 1, it is straightforward to know that if $n \leq m$, $\lambda_i \leq \beta_i$, for $i = 1, \ldots, t_2$.

It is important to note that the simplest way to establish the dominance relationships between vertices in H_m and q_m is to set both n and m to 1, that is, $HEE(H_m, v_j, 1)$ and $HEE(q_m, u_i, 1)$. However, this type of embedding only considers first-order structural information and loses the multi-order structural information of the vertex. Therefore, unless the query hypergraph is very small, n is generally set to be greater than 1. Furthermore, since we aim to compute the embedding of vertices in the hypergraph-mapping graph only once to accommodate hypergraph matching problems with different numbers of hyperedges, we set m to be greater than n.We add very small values to the end of $HEE(q_m, u_i, n)$ to make its length equal to that of $HEE(H_m, v_j, m)$.

Definition 10 (Vector Dominance). *Given two vectors $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, A is said to be dominated by B if it satisfies:*

1. $\forall 1 \leq i \leq n$, $a_i \leq b_i$.

2. $\exists 1 \leq j \leq n$, $a_j < b_j$.

It is clear that if vertex v_j matches with vertex u_i, then $HEE(q_m, u_i, n)$ is definitely dominated by $HEE(H_m, v_j, m)$.

Definition 11 (Hyperedge Signature Embedding). *In the hypergraph mapping graph G_m, the hyperedge signature embedding of a vertex v is the one-hot encoding of its vertex label, corresponding to the hyperedge signature of the hyperedge associated with v, denoted as $HSE(v)$.*

For example, in Figure 2, we encode the vertex label $[A, A, B, C]$ as $[0, 0, 1, 0]$, so $HSE(u_1) = [0, 0, 1, 0]$. If a vertex v_j in the data hypergraph-mapping graph H_m matches with a vertex u_i in the query graph q_m, then $HSE(u_i)$ is equal to $HSE(v_j)$.

Based on hyperedge eigenvalue embedding, we can obtain multidimensional structural information of the hyperedges. Through hyperedge signature embedding, we can acquire information about the vertices within the hyperedges. By utilizing these two embeddings, we can filter out many hyperedges in the data hypergraph that are irrelevant to the query. However, since we let $m > n$ in $HEE(q_m, u_i, n)$ and $HEE(H_m, v_j, m)$, the candidate set is filtered by HEE and HSE may have false positives.

For example, in the hypergraph-mapping graph of Figure 2, $HEE(q_m, u_1, 1)$ is dominated by $HEE(H_m, v_4, 2)$, $HSE(u_1)$ is equal to $HSE(v_4)$, but vertex u_1 does not match with vertex v_4.

In order to remove such false positives as much as possible, we propose hyperedge neighbor embedding.

Definition 12 (Hyperedge Neighbor Embedding). *In the hypergraph mapping graph G_m, the hyperedge neighbor embedding $HNE(v)$ of vertex v is the concatenation of the sum of the one-hot encoding of the labels of the edges connected to vertex v, denoted as $HOVE(v)$, and the sum of the hyperedge signature embeddings of the neighbor vertices of v, denoted as $HNSE(v)$.*

Theorem 3. *Given a data hypergraph H and a query hypergraph q. If a vertex v_j in the data hypergraph-mapping graph H_m matches with a vertex u_i in the query graph q_m, it must satisfy that $\forall HNE(v_j)[l] \geq HNE(u_i)[l]$, for $0 \leq l \leq |HNE(v_j)| - 1$.*

Proof. Since vertex v_j in the data hypergraph-mapping graph H_m matches with vertex u_i in the query graph q_m, $\forall u_k \in N(u_i)$, it must be that $\exists f(u_k) \in N(v_j)$. Therefore, each element value in $HNSE(v_j)$ is greater than $HNSE(u_i)$. And $\forall (u_k, u_i) \in E(q_m)$, it must be that $\exists (f(u_k), v_j) \in E(H_m)$. Consequently, each element value in $HOVE(v_j)$ is greater than $HOVE(u_i)$. In summary, each element value in $HNE(v_j)$ is greater than $HNE(u_i)$.

Definition 13 (Hyperedge Embedding). *In the hypergraph-mapping graph G_m, the hyperedge embedding of a vertex v is obtained by concatenating $HEE(v)$, $HSE(v)$, and $HNE(v)$, denoted as $HE(v) = HEE(v) \oplus HSE(v) \oplus HNE(v)$.*

Theorem 4. *Given a data hypergraph H and a query hypergraph q. If a vertex v_j in the data hypergraph-mapping graph H_m matches with a vertex u_i in the hypergraph-mapping graph q_m, it must satisfy that $\forall HE(v_j)[l] \geq HE(u_i)[l]$, for $0 \leq l \leq |HE(v_j)| - 1$, that is, $HE(v_j)$ must dominate $HE(u_i)$.*

Proof. If a vertex v_j in the data hypergraph-mapping graph H_m matches with a vertex u_i in the query graph q_m, then $\forall HEE(v_j)[l] \geq HEE(u_i)[l]$, $\forall HSE(v_j)[l] = HSE(u_i)[l]$, and $\forall HNE(v_j)[l] \geq HNE(u_i)[l]$. Therefore, $\forall HE(v_j)[l] \geq HE(u_i)[l]$.

4 The HESM Method

4.1 Hyperedge Embedding Index

To efficiently query hyperedge embeddings, we utilize an R-tree, a widely used spatial index for managing multidimensional data. R-trees make spatial queries more efficient by organizing spatial objects into overlapping rectangular boundaries, which are represented as nodes. We build an R-tree index on data hyperedge embeddings.

Leaf Nodes. Each leaf node $N \in I_j$ contains multiple vertices v_j, where each vertex has a hyperedge embedding vector $HE(v_j)$.

Non-Leaf Nodes. Each non-leaf node $N \in I_j$ contains multiple entries N_i, each of which is an MBR, N_i.MBR, of all hyperedge embeddings $HE(v_j)$ under entry N_i.

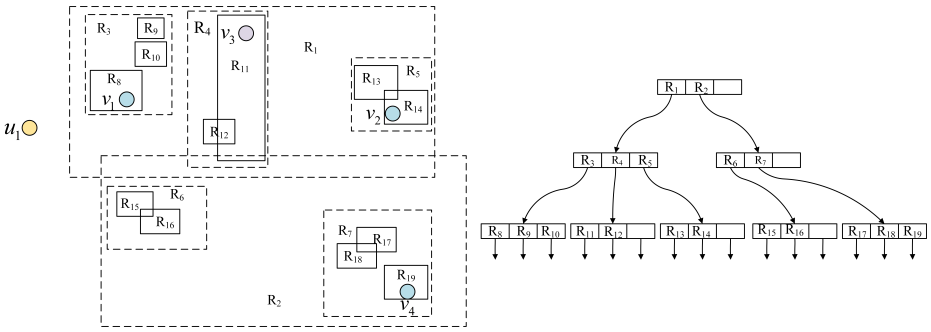

Fig. 3. Example of R-tree index

Each node in an R-tree corresponds to a region within the data space, known as a *Minimal Bounding Rectangle (MBR)*. Through the MBR structure, the R-tree can effectively and efficiently determine whether a queried data object might exist within the managed area of a node, thereby excluding nodes that do not contain the queried data object as early as possible during the query process.

As illustrated in Fig. 3, we build an R-tree index for the hyperedge embeddings of vertices in the data hypergraph-mapping graph. This allows us to quickly find the candidate set during queries using the R-tree.Additionally, by incorporating multi-dimensional structural information, we can more efficiently

and accurately filter out vertices that are unrelated to the query. When the hyperedge embedding vectors are particularly long, we build the index solely on the hyperedge signature embeddings. In this case, we let $m = n$ in $HEE(q_m, u_i, n)$ and $HEE(H_m, v_j, m)$.

4.2 Subhypergraph Matching Algorithm

Filtering. The main goal of the filtering stage is to use some features or attributes to quickly identify vertices in the data graph that may correspond to the query graph, thereby excluding those graph vertices or edges that are unlikely to match the query subgraph and significantly reducing the number of candidate subgraphs to be verified.

We use the ranking function: $\text{Rank}(u_i) = \text{freq}(H_m, L(u_i))/\text{degree}(u_i)$ to determine the generation order of the candidate set of vertices in the query hypergraph-mapping graph q_m, where $\text{freq}(H_m, L(u_i))$ is the number of times the label of vertex u_i in the query hypergraph-mapping graph q_m appears in the data hypergraph-mapping graph H_m, and $\text{degree}(u_i)$ is the degree of u_i. We construct the candidate set starting from the vertex with the smallest ranking function value.Then, We use the hyperedge embedding index to find the candidate set of vertices in the query mapping graph q_m.

Algorithm 1: R-tree Filtering Algorithm

Input : hyperedge embedding index, query hyperedge embedding $HE(u)$, and generation order π
Output: Candidate set C

```
1  for u_i ∈ π do
2      M ← R-tree.root;
3      N ← R_Search(M, u);
4      Insert ∀v_j ∈ N into C(u_i);
5  Function R_Search(M, u):
6      if M is a leaf node then
7          return DR(HE(u_i)) ∩ M.MBR;
8      else
9          for i ← 1 to M.count do
10             if DR(HE(u_i)) ∩ M_i.MBR ≠ ∅ then
11                 R_Search(M.CP_i, u);
```

Algorithm 1 is the filtering algorithm based on the R-tree. We follow the candidate set generation order and use function `R_Search` to find the dominating MBR in the R-tree for the hyperedge embedding of each vertex u_i in the query hypergraph-mapping graph. All data hyperedge embeddings contained within this MBR constitute the candidate set for u_i (Lines 1-4). Function `R_Search`

recursively checks whether the lower bounds of all dimensions of the MBRs for a node or its child nodes in the R-tree are greater than or equal to the corresponding dimensional values in u_i (Lines 5-11). After finding the candidate set of vertices, we can find the candidate set of edges by using the connection relationships between the vertices.

Parallel Join Strategy. We propose an edge-based parallel join strategy that, compared to vertex-based parallel methods, better leverages the parallel capabilities of GPUs when the query graph and data graph are relatively small.

Definition 14 (Edge Spanning Tree). *Starting from the initial vertex u_s, according to the depth-first traversal strategy, the query edges visited during the traversal are used as paths in the spanning tree until all query edges are visited and the traversal process ends. The resulting spanning tree is called an edge spanning tree, denoted by ET_Q.*

To minimize the height of the spanning tree and thus reduce the overhead during parallel joins, we select the vertex u_s with the highest degree in the query hypergraph-mapping graph q_m as the initial vertex for the edge spanning tree.

The edge-spanning tree of the query hypergraph-mapping graph q_m from Fig. 4(a) is shown in Fig. 4(b).

(a) query hypergraph mapping graph (b) edge spanning tree

Fig. 4. Example of edge spanning tree

The query hypergraph-mapping graph is split into branches based on its edge-spanning tree, and each branch is processed independently to obtain its edge candidate set. The candidate sets can be processed in parallel using GPUs. The matching result set M is formed by connecting the candidate vertices at the intersection of the branches.

The parallel branch splitting process of the edge-spanning tree ET_{q_m} in Fig. 4(b) is shown in Fig. 5. ET_{q_m} contains three branches: $u_2 - u_0 - u_1$, $u_2 - u_1$, and $u_2 - u_3$. Branch 1 includes two query edges, $e(u_0, u_1)$ and $e(u_0, u_2)$. Using the GPU to process the three branches in parallel, we find the candidate results for each branch.

For Branch 1, the candidate results of $e(u_0, u_1)$ and $e(u_0, u_2)$ need to be joined to obtain the candidate results for Branch 1, while the other two branches only

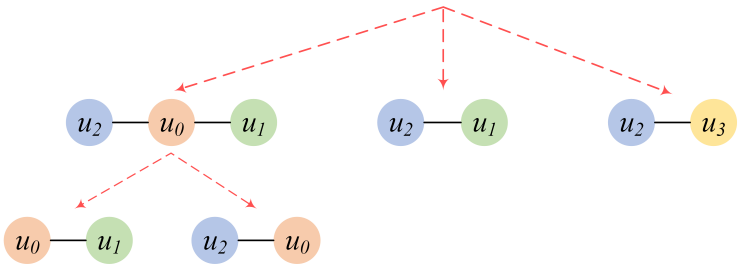

Fig. 5. Example of branch splitting process

require finding the candidates for their respective edges. By joining the candidate vertices of the intersection u_2 of the three branches, we obtain the final matching results.

Algorithm 2: Parallel Subgraph Matching Algorithm

Input: Edge spanning tree ET and edge candidate set Cand_edge
Output: Subgraph matching result M

```
1 ET_branch ← Spanningtree_split(ET);
2 M_branch ← ∅;
3 foreach branch ∈ ET_branch do
4     edge_branch ← Get_branch_edge(branch);
5     foreach edge ∈ edge_branch do
6         edge_cand ← GetCandidate(Cand_edge, edge, warp_id);
7         Joint_edege_candidate(M_branch[branch], edge_cand);
8 M ← Joint_branch_candidate(M_branch, warp_id);
```

Algorithm 2 is a parallel subgraph matching algorithm based on edge connections. Initially, we apply the `Spanningtree_split` function to branch-split the edge-spanning tree ET, resulting in independent tree branches (Line 1). We then utilize GPU parallelism to enumerate connections in parallel for each tree branch, generating candidate match results for the branches. The `Get_branch_edge` function is used to process edge splitting for each branch and record connection point information, producing branch edges edge_branch. Each thread group, via the `Get_branch_edge` function, is tasked with finding candidate edge sets for each branch query edge. Following this, the `Joint_edege_candidate` function aggregates and connects candidate edges based on branch connection points, identifying candidate matches for each tree branch (Lines 3-7).

Finally, the `Joint_branch_candidate` function aggregates candidate matches for each branch through branch connection points. The parallel processing improves aggregation efficiency, generating the final subgraph matching results (Line 8).

4.3 Subhypergraph Matching Algorithm Based on Hyperedge Embedding

Algorithm 3 is the subhypergraph matching algorithm based on hyperedge embedding. The algorithm is divided into two phases: an offline phase and an online phase. The offline phase first converts the hypergraph H into a hypergraph-mapping graph H_m (Line 2), then calculates the hyperedge embedding of each vertex in H_m (Lines 3-4), and finally builds an R-tree index on the hyperedge embedding (Line 5). In the online phase, the query hypergraph q is first converted into the query hypergraph-mapping graph q_m (Line 7), then the embedding of each vertex in q_m is calculated (Lines 8-9), and the R-tree filtering algorithm is applied to find the candidate set for each vertex (Line 10). Finally, the parallel subgraph matching algorithm is used to determine the final subhypergraph matching result (Line 11).

Algorithm 3: Subhypergraph Matching Algorithm Based on Hyperedge Embedding

Input: A query hypergraph q and a data hypergraph H
Output: Subgraph matches M
1 // **Offline Pre-Computation Phase**
2 Map H to the hypergraph-mapping graph H_m;
3 **foreach** *vertex* v_j *in* H_m **do**
4 Compute $HE(v_j)$;
5 Build an R-tree index on the hyperedge embeddings;
6 // **Online Subhypergraph Matching Phase**
7 Map q to the hypergraph-mapping graph q_m;
8 **foreach** *vertex* u_i *in* q_m **do**
9 Compute $HE(u_i)$;
10 $C(u_i) \leftarrow$ Generate candidate vertex sets using the ***R-tree Filtering Algorithm***;
11 Use the ***Parallel Subgraph Matching Algorithm*** to obtain the matching results;

5 Experiments

5.1 Experimental Setup

Baselines. We compare the subhypergraph matching algorithm proposed in this paper with the most advanced subgraph matching algorithms: CFL, CECI, GQL, QuickSIM [17], and an Ullman-based subhypergraph matching algorithm. In particular, we have modified the subgraph matching algorithms to make them suitable for hypergraphs. Since these comparison algorithms are all serial, we use single threading for comparison.

Datasets. The experiments are conducted on six real-world hypergraph datasets with labeled vertices, downloaded from [18]. The descriptions of these

Table 1. Summary of datasets.

Dataset	vertices	hyperedges	classes	mean hyperedge	max hyperedge
house-committees	1290	341	2	34.8	82
senate-committees	282	315	2	17.2	31
contact-high	327	7,818	9	2.3	2
mathoverflow-answers	73,851	5,446	1,456	24.2	1,784
senate-bills	294	29,157	2	8	99
house-bill	1,494	60,987	2	20.5	399

data hypergraphs are shown in Table 1. We extract m connected hyperedges from these data hypergraphs as our query graph, to ensure that there will always be matching subhypergraphs in the data hypergraphs. To ensure the accuracy of the experiment, we take the average of twenty query hypergraphs as the experimental result.

5.2 Experimental Analysis

Subhypergraph Matching Algorithm Comparison. We first compare our algorithm with other algorithms on labeled hypergraph datasets.The experimental results are shown in Fig. 6.

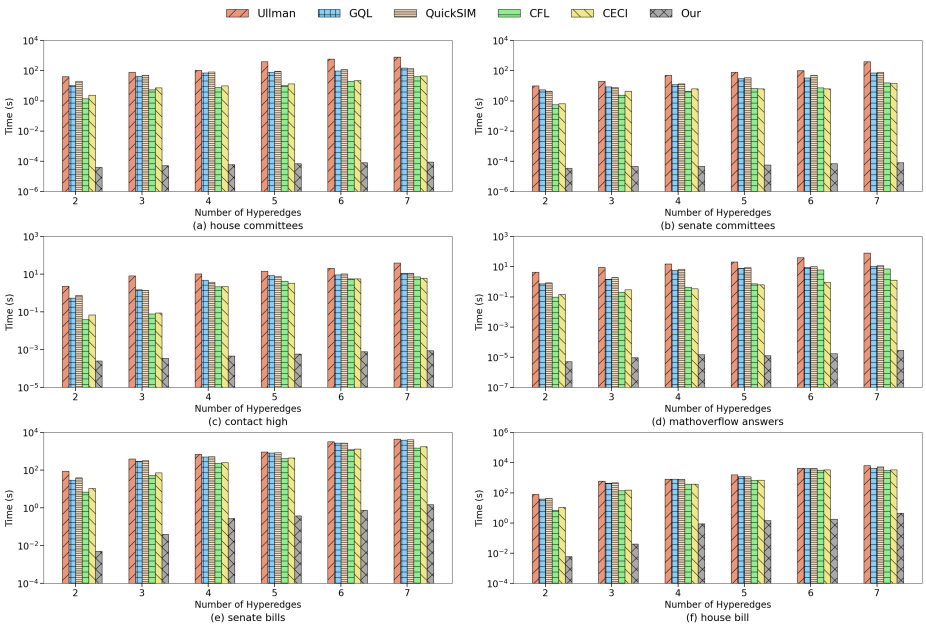

Fig. 6. Comparison of subhypergraph matching time

The results indicate that the subhypergraph matching algorithm we proposed significantly reduces the query time. This is because we consider the information of hyperedges and the information between hyperedges. In the datasets *senate-bills* and *house-bill*,the number of hyperedges far exceeds the number of vertices, and there is a significant amount of vertex sharing between hyperedges. This lead to bloat when converted to a hypergraph map. However, the hypergraph embedding can effectively filter out vertices that are irrelevant to the query. Therefore, it has almost no impact on query performance.

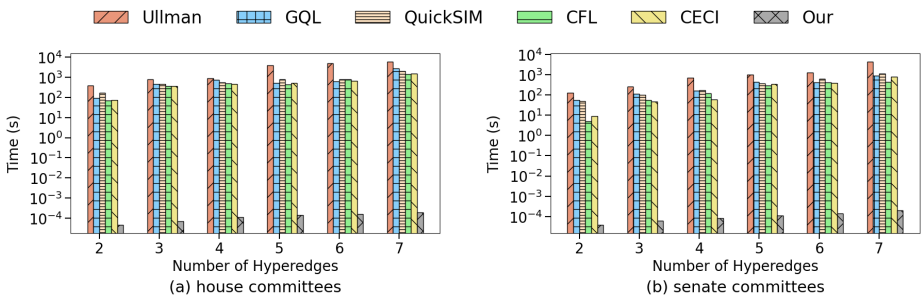

Fig. 7. Comparison of unlabeled subhypergraph matching time

Comparison of Algorithms on Unlabeled Datasets. We then compare our algorithm with other algorithms on unlabeled datasets. The datasets used are the *house-committees* and *senate-committees* datasets with vertex labels removed.

The experimental results are shown in Fig. 7. From the results, it can be seen that the query time increases sharply for other algorithms because they cannot effectively filter the vertices in the query hypergraph using vertex labels. Our proposed hypergraph-mapping graph transformation method adds labels to unlabeled hypergraphs, which allows for the effective filtering of irrelevant vertices. As a result, the query time does not increase significantly.

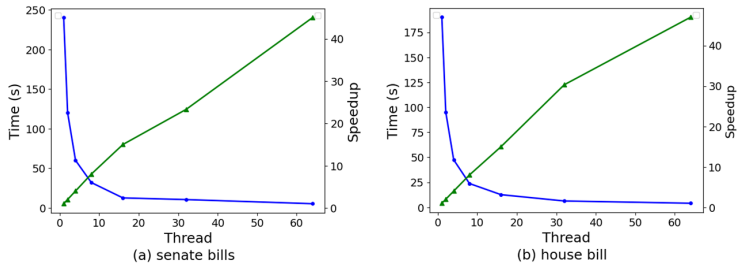

Fig. 8. Time for different number of threads

Parallel Analysis. We analyzed the parallel performance of the subhypergraph matching algorithm using query hypergraphs with 9 hyperedges on the *senate-bills* and *house-bill* datasets.

The experimental results are shown in Fig. 8. It can be seen from the experimental results that our parallel subhypergraph matching algorithm significantly reduces query time and improves query efficiency.

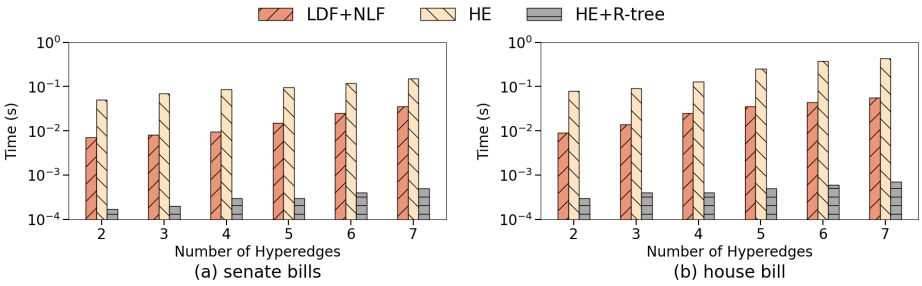

Fig. 9. Comparison of filtering time

Filtering Time of Hyperedge Embedding Index. We finally verify how much time our hyperedge embedding can save in the filtering stage. The datasets used are the *senate-bills* and *house-bill* datasets.

The experimental results are shown in Fig. 9. It can be seen from the experimental results that although our hyperedge embedding method avoids traversing the data hypergraph during filtering, directly comparing each query hyperedge embedding with all data hyperedge embeddings would result in significant time consumption, even far exceeding the time required for label filtering and neighbor label filtering. However, by creating an index on the hyperedge embeddings using an R-tree, we can leverage the R-tree's efficient querying capabilities to significantly accelerate the filtering process.

6 Conclusion

In this paper, we study an NP-complete problem, subhypergraph isomorphism query. We introduce a lossless transformation structure for hypergraphs called the hypergraph-mapping graph. Based on this structure, we propose a hyperedge embedding method that ensures that if there is a matching relationship between a hyperedge in the data hypergraph and a hyperedge in the query hypergraph, the embedding of the data hyperedge dominates the embedding of the matching query hyperedge. Subsequently, we employ spatial indexing methods to build indices for the hyperedge embeddings of the data hypergraph. In this way, we can directly use the index to quickly and accurately find data hyperedges that may have a matching relationship with the query hyperedge without needing to traverse the entire data hypergraph. Finally, we propose an edge-join parallel

algorithm to improve the efficiency of subgraph matching. Experimental results show that our proposed method outperforms other methods.

References

1. L. Hong, L. Zou, X. Lian, S.Y. Philip: Subgraph matching with set similarity in a large graph database. IEEE TKDE, 27(9), 2507–2521 (2015)
2. J. Llados, E. Marti, J.J. Villanueva: Symbol recognition by error-tolerant sub-graph matching between region adjacency graphs. IEEE TPAMI 23(10), 1137–1143 (2001)
3. D. Conte, P. Foggia, C. Sansone, M. Vento: Thirty years of graph matching in pattern recognition. Int. J. Pattern Recognit. Artif Intell. 18(03), 265–298 (2004)
4. Hwang T H, Tian Z, Kuangy R, et al. Learning on weighted hypergraphs to integrate protein interactions and gene expressions for cancer outcome prediction[C]//2008 Eighth IEEE ICDE. IEEE, 2008: 293-302
5. Ramadan E, Tarafdar A, Pothen A. A hypergraph model for the yeast protein complex network[C]//18th IPDPS, 2004. Proceedings. IEEE, 2004: 189
6. Feng S, Heath E, Jefferson B, et al. Hypergraph models of biological networks to identify genes critical to pathogenic viral[J]. system, 2021, 8: 10
7. Klamt S, Haus U U, Theis F. Hypergraphs and cellular networks[J]. PLoS computational biology (2009), 1–6
8. Hwang T H, Tian Z, Kuangy R, et al. Learning on weighted hypergraphs to integrate protein interactions and gene expressions for cancer outcome prediction[C]//2008 Eighth IEEE ICDE. IEEE, 2008: 293-302
9. Loc Tran. 2012. Hypergraph and protein function prediction with gene expression data. (2012)
10. Su Y., Gu Y., Wang Z. et al.: Efficient subgraph matching based on hyperedge features. IEEE Transactions on Knowledge and Data Engineering, (2022)
11. Ha T.W., Seo J.H., Kim M.H.: Efficient searching of subgraph isomorphism in hypergraph databases. In: 2018 IEEE BigComp. IEEE, pp. 739–742 (2018)
12. Su Y., Gu Y., Wang Z. et al.: Efficient subgraph matching based on hyperedge features. IEEE TKDE, (2022)
13. Yang Z., Zhang W., Lin X. et al.: HGMatch: A Match-by-Hyperedge Approach for Subgraph Matching on Hypergraphs. In: 2023 ICDE. IEEE, pp. 2063–2076 (2023)
14. Shi T., Zhai M., Xu Y. et al.: Graphpi: High performance graph pattern matching through effective redundancy elimination. In: International Conference for High Performance Computing, Networking. IEEE, pp. 1–14 (2020)
15. F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient Subgraph Matching by Postponing Cartesian Products," in SIGMOD'16. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 1199–1214
16. D. B. West. Introduction to Graph Theory (2nd Edition). Prentice Hall, 2000
17. S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in Proceedings of SIGMOD, 2020, pp. 1083–1098
18. A. R. Benson. Hypergraph datasets. https://www.cs.cornell.edu/arb/data/, 2022