CEMR: An Effective Subgraph Matching Algorithm with Redundant Extension Elimination

Linglin Yang*
Peking University
Beijing, China
linglinyang@stu.pku.edu.cn

Xunbin Su*
Peking University
Beijing, China
suxunbin@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

Xiangyang Gou
University of New South Wales
Sydney, Australia
xiangyang.gou@unsw.edu.au

Yinnian Lin
Peking University
Beijing, China
linyinnian@pku.edu.cn

ABSTRACT

Subgraph matching is a fundamental problem in graph analysis with a wide range of applications. However, due to its inherent NP-hardness, enumerating subgraph matches efficiently on large real-world graphs remains highly challenging. Most existing works adopt a depth-first search (DFS) backtracking strategy, where a partial embedding is gradually extended in a DFS manner along a branch of the search trees until either a full embedding is found or no further extension is possible. A major limitation of this paradigm is the significant amount of duplicate computation that occurs during enumeration, which increases the overall runtime. To overcome this limitation, we propose a novel subgraph matching algorithm, CEMR. It incorporates two techniques to reduce duplicate extensions: *common extension merging*, which leverages a black-white vertex encoding, and *common extension reusing*, which employs common extension buffers. In addition, we design two pruning techniques to discard unpromising search branches. Extensive experiments on real-world datasets and diverse query workloads demonstrate that CEMR outperforms state-of-the-art subgraph matching methods.

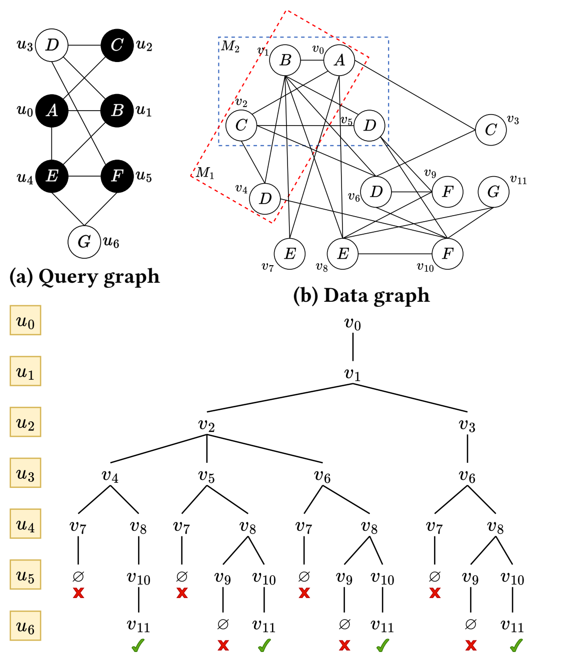
PVLDB Reference Format:
Linglin Yang, Xunbin Su, Lei Zou, Xiangyang Gou, and Yinnian Lin. CEMR: An Effective Subgraph Matching Algorithm with Redundant Extension Elimination. PVLDB, 19(7): 1572 - 1585, 2026.
doi:10.14778/3801059.3801070

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/pkumod/CEMR.

1 INTRODUCTION

In recent years, graph structures have gained increasing significance in the data management community. As one of the fundamental tasks in graph analysis, subgraph matching has attracted much attention from both industry and academia [27, 29, 33] and is widely used in many applications, such as chemical compound search [42], social network analysis [11], protein-protein interaction network analysis [24] and RDF query processing [23, 50].

Given a data graph G and a query graph Q, subgraph matching (isomorphism) aims to find all the subgraphs of G that are isomorphic to Q. The mapping between each of these subgraphs and Q is called an embedding (or match) of Q over G. For example, given a data graph G and a query graph Q in Figure 1, $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_5), (u_4, v_8), (u_5, v_{10}), (u_6, v_{11})\}$ is one embedding of Q over G. Unfortunately, subgraph matching is a well-known NP-hard problem [13] and the data graph G is usually very large in practical applications. Thus, it is challenging to efficiently enumerate all embeddings of Q over G.



(a) Query graph (b) Data graph

(c) Search tree rooted at $\{(u_0, v_0)\}$

Figure 1: Subgraph Matching

Many algorithms [5, 9, 16, 17, 20, 22, 32, 34, 39, 46, 48] have been proposed to solve the subgraph matching problem. A typical subgraph matching algorithm framework finds matches of the query

*Linglin Yang and Xunbin Su contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801070

graph Q over the data graph G by iteratively mapping each query vertex in Q to data vertices in G following a certain matching order. Such a matching process can be organized as a backtracking search on a search tree.

For example, Figure 1c shows a search tree when mapping u_0 to v_0 from the previous example. Each tree node at level i ($i \in \{0, \ldots, |V(Q)| - 1\}$) represents an embedding M that matches the first $i + 1$ query vertices in the matching order. A green tick denotes a valid embedding, whereas a red cross denotes that the corresponding tree node cannot produce a valid embedding. We call an embedding at level $|V(Q)| - 1$ *full embedding* and an embedding at level less than $|V(Q)| - 1$ *partial embedding*. By adding matches of u_{i+1} with different data vertices into a partial embedding M at level i, we can extend M to its children at level $i + 1$.

Most existing solutions adopt the *depth-first search (DFS)* backtracking approach. They gradually extend one partial embedding following a certain branch in the search tree (i.e., perform DFS on the search tree) until a full embedding is found or no extension can be made. Then they backtrack to the upper level and try to search another branch. To speed up subgraph matching, existing solutions have proposed many heuristic optimization techniques on getting good matching orders [5, 16, 21, 36], generating the smaller candidate space [4, 5, 16, 17, 39], sharing computation among query vertices with equivalence relationship [17, 22, 32] and reducing duplicated computation [2, 14, 15, 19, 20, 31, 41, 45]. Some of the remaining works [1, 25, 35] adopt the *breadth-first search (BFS)* approach where they find all partial embeddings at the same level of the search tree and then move on to the next level. *BFS-based* solutions suffer from a large amount of memory consumption for storing intermediate results [7, 20, 40]. Furthermore, in cases where a query cannot be completed within reasonable time, *DFS-based* methods are capable of obtaining a portion of the final embeddings within the given time constraint, whereas *BFS-based* methods may yield no results at all. In this paper, we focus on optimizing *DFS-based* approaches.

1.1 Motivation

During the enumeration phase, there may exist duplicate extension computations. These duplicate computations usually occur at the same level of the search tree, where the backward neighbors of the next query vertex u share the same mappings across different partial embeddings. The root cause is that the extension of u depends only on its backward neighbors and is independent of other query vertices. These repeated calculations enlarge the search space and hinder enumeration performance. We use the following example to illustrate this phenomenon.

Example 1. *Consider the data graph and query graph in Figure 1, with matching order $O = (u_0, u_1, u_2, u_3, u_4, u_5, u_6)$. Examine two partial embeddings: $M_1 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_4)\}$ (red dashed box) and $M_2 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_5)\}$ (blue dashed box). Both share the prefix $(u_0, u_1) \mapsto (v_0, v_1)$. Notably, when extending to u_4, M_1 and M_2 perform the same extension and generate the same candidate set $\{v_7, v_8\}$, since u_4 only connects to u_0 and u_1. This redundancy implies that extending M_2 could reuse the results from M_1, allowing simultaneous or shared extension and avoiding repeated computation.*

In BFS-based enumeration, it is straightforward to eliminate such redundancy by grouping partial embeddings that share the same extension patterns at the same level of the search tree, since a complete intermediate result table can be constructed in each iteration. However, the memory cost of BFS can be prohibitively high. In contrast, when using DFS for enumeration, it becomes challenging to share common extension results among partial embeddings, as the repeated extensions may appear in different branches of the search tree. This necessitates a mechanism to reduce duplicate extension computations during the backtracking process.

1.2 Our Solution

To eliminate computation redundancy, we propose two core optimization techniques: *common extension merging* (`CEM`) and *common extension reusing* (`CER`), which together form our method `CEMR` (**C**ommon **E**xtension **M**erge and **R**eusing).

`CEM` enables joint extension of multiple search branches by merging them until divergence. To support this, we introduce a *black-white vertex encoding* scheme (Section 4), which partitions query vertices into black and white. A white vertex can match multiple data vertices within a single partial embedding, e.g., treating u_4 as white resolves the issue illustrated in Example 1.

`CER` leverages *common extension buffers* (CEBs, see Section 5.2) to cache reusable extension results that can be shared across multiple partial embeddings, thereby reducing redundant computation.

In addition, we design two pruning techniques that efficiently identify and discard unpromising search branches during the backtracking enumeration process.

To summarize, we make the following contributions in this paper.

- We propose a DFS-based subgraph matching algorithm, `CEMR`, that aims to reduce redundant computation during the enumeration phase.
- We develop a forward-looking common extension merging technique based on black-white vertex encoding, which effectively merges search branches with similar expansion behaviors. In addition, we propose a cost-driven encoding strategy designed to maximize computational cost reduction.
- We propose a backward-looking common extension reusing technique that caches and reuses extension results to avoid repeated computation.
- We introduce two pruning techniques that effectively eliminate unpromising search branches and redundant partial embeddings.
- We conduct extensive experiments on several real-world graph datasets. Results show that `CEMR` consistently outperforms state-of-the-art algorithms.

2 PRELIMINARY AND RELATED WORKS

2.1 Problem Definition

In this paper, we focus on undirected, vertex-labeled, and connected graphs. Note that our techniques can be extended to more general cases (e.g., directed graphs and edge-labeled graphs, we provide some discussions on these extensions in Section 6.4).

We use G and Q to denote the data graph and the query graph, respectively. A (data) graph G is a quadruple $G = (V, E, \Sigma, L_G)$, where $V(G)$ is the set of vertices, $E(G) \subseteq V(G) \times V(G)$ is the set of edges, Σ is the set of labels, and L_G is a labeling function that assigns a label from Σ to each vertex in V. Given $v \in V(G)$, $N(v)$ denotes the set of neighbors of v, i.e., $N(v) = \{v' \mid e(v,v') \in E(G)\}$. The degree of v, denoted by $d(v)$, is defined as $d(v) = |N(v)|$. The query graph Q is defined in the same way.

Definition 2.1 (Subgraph Isomorphism). Given a query graph Q and a data graph G, Q is subgraph isomorphic to G if there exists an injective mapping function M from $V(Q)$ to $V(G)$ such that:

- (Label constraint) $\forall u \in V(Q), L_Q(u) = L_G(M[u])$;
- (Topology constraint) $\forall e(u,u') \in E(Q), e(M[u], M[u']) \in E(G)$.

Sometimes, the mapping function M is also called an embedding.

Definition 2.2 (Subgraph Matching Problem). Given a query graph Q and a data graph G, the subgraph matching problem is to find all distinct subgraphs (*embeddings*) of G that are *isomorphic* to Q.

An embedding of an induced subgraph of Q in G is called a *partial embedding*. In contrast, an embedding of the complete query graph Q is referred to as a *full embedding*.

Definition 2.3 (Matching Order). A matching order is a permutation of the query vertices, denoted as $O = (u_0, u_1, \ldots, u_{n-1})$, where we assume $|V(Q)| = n$ throughout the rest of the paper. A valid matching order must preserve the connectivity of the query graph. That is, for each $i \in \{1, \ldots, n-1\}$, the subquery Q_i induced by the first i vertices $\{u_0, u_1, \ldots, u_{i-1}\}$ in O must be connected. For clarity, we consistently use this notation throughout the paper.

Definition 2.4 (Backward (Forward) Neighbors). Given a matching order O, the set of backward neighbors $N_-^O(u)$ (forward neighbors $N_+^O(u)$) of a query vertex u is the set of neighbors of u whose indices in O are smaller (greater) than u's corresponding index $o(u)$ in O. Formally, $N_-^O(u_i) = \{u \mid u_j \in N(u), j < i\}$ and $N_+^O(u_i) = \{u_j \mid u_j \in N(u), j > i\}$.

Example 2. *Considering the query graph Q in Figure 1a, assume that the matching order is $O = (u_0, u_1, u_2, u_3, u_4, u_5, u_6)$. For u_4, it has four neighbors u_0, u_1, u_5, and u_6. Among them, u_0 and u_1 are matched before u_4, which are backward neighbors of u_4 (i.e., $N_-^O(u_4) = \{u_0, u_1\}$). In contrast, u_5 and u_6 are forward neighbors of u_4 (i.e., $N_+^O(u_4) = \{u_5, u_6\}$).*

Table 1 lists the frequently used notations in this paper.

Table 1: Frequently used notations.

Notation	Description
G, Q	Data graph and query graph
v, u	Data vertex and query vertex
O	Matching order
Q_i	Induced subquery by the first i vertices
$N_-^O(u), N_+^O(u)$	Backward and forward neighbors of u given O
$BK(u), WT(u)$	Black and white backward neighbors of u
$M, M[u]$	A (partial) embedding , mapping of u in M
$C(u)$	Candidate vertex set of u
$\mathcal{A}$	Auxiliary data structure
$\mathcal{A}^u_{u'}(v)$	Neighbors of v in $C(u')$ where $v \in C(u)$
$R_M(u)$	Extensible vertices of u under M
$RS(u)$	Reference set of u
$Con(u)$	Contained vertex set of u

Algorithm 1: Generic framework of *indexing-enumeration*

Input: Query graph Q, data graph G.
Output: All embeddings of Q in G, denoted as $\mathcal{M}$.

```
1 C, 𝒜 ← generate candidate vertices and an auxiliary index
2 O ← get the matching order
3 ℳ ← ∅
4 Enumerate(Q, 𝒜, O, ℳ, {}, 0)
5 return ℳ
```

2.2 Background and Related Work

Most recent studies [3–5, 8, 16, 17, 22, 28, 39] on subgraph matching are based on the *preprocessing-enumeration* framework [37] as shown in Algorithm 1. They first find candidates of query vertices and edges to build an online auxiliary structure, then enumerate all embeddings with the help of this auxiliary structure.

2.2.1 Preprocessing Phase. In Algorithm 1, the preprocessing phase consists of filtering and matching order generation. Specifically, it first generates a candidate vertex set C ($C(u)$ for each $u \in V(Q)$) and builds an auxiliary data structure $\mathcal{A}$ to maintain the candidate edges between the candidate vertex sets (line 1). Then, the algorithm generates a matching order O usually in a greedy way based on some heuristic rules (line 2).

2.2.2 Enumeration Phase. After the preprocessing phase, Algorithm 1 recursively enumerates matches along the matching order O (line 4). The enumeration procedure is detailed in Algorithm 2, which consists of two key steps. Given the current partial embedding M, line 3 computes all extensible vertices $R_M(u_i)$ for the next query vertex u_i. Then, lines 4-5 recursively extend the partial embedding by exploring all candidates in $R_M(u_i)$.

Algorithm 2: Enumerate(Q, $\mathcal{A}$, O, $\mathcal{M}$, M, i) (Basic)

Input: The query Q, auxiliary structure $\mathcal{A}$, matching order O, result set $\mathcal{M}$, an embedding M of Q_i, and the backtracking depth i.

```
1 if i = |V(Q)| then
2 |  ℳ ← ℳ ∪ {M}, return
3 R_M(u_i) = ∩_{u ∈ N_-^O(u_i)} 𝒜^u_{u_i}(M[u])
4 foreach v ∈ R_M(u_i) do
5 |  Enumerate(Q, 𝒜, O, ℳ, M ∪ {(u_i, v)}, i + 1)
```

2.2.3 Related Works of Preprocessing Phase. The label degree filter (LDF) and neighbor label filter (NLF) [49] are the two most widely used filtering methods. They check the label and degree of a data vertex and its neighborhood to eliminate vertices that cannot be matched to a query vertex. In addition to vertex candidates, CFL [5] also builds an auxiliary structure to maintain the edge candidates of a BFS tree in the query graph, enabling more powerful filtering. CECI [4] and DAF [16] further consider the non-tree edges of the query graph in their auxiliary structures. RM [39] utilizes the semi-join operator to quickly eliminate dangling tuples. VC [38] and VEQ [22] adopt more sophisticated filtering rules to produce tighter candidate sets, at the cost of increased filtering time.

Most existing subgraph matching methods use heuristic vertex ordering strategies. RI [6] and RM [39] determine the query vertex order solely based on the structure of the query graph, prioritizing vertices from its dense regions. Other methods adopt different heuristics, such as selecting vertices with higher degrees and smaller candidate sets [5, 18]. DAF [16] and VEQ [22] further propose adaptive ordering strategies, where the choice of the next query vertex may vary across different partial embeddings.

2.2.4 Related Works of Enumeration Phase. As the enumeration phase is usually the most time-consuming part of subgraph matching, many works have proposed various optimizations to accelerate it, mainly by reducing redundant computation. These optimizations can be broadly classified into two categories. The first is backward-looking optimization [3, 16, 22, 39], which caches intermediate results to avoid redundant extensions. The second is forward-looking optimization [20, 26, 44], which merges multiple search branches and extends them simultaneously in subsequent steps.

Backward-Looking Optimization. Backward-looking optimization leverages historical extensions to accelerate subsequent ones. It typically serves as a pruning technique to discard unpromising search branches based on past extension results.

DAF [16] proposes a *failing set* pruning strategy to eliminate unpromising search branches. The basic idea is: Given a partial embedding M, if M cannot be extended to form a complete match of Q, some other search branches (satisfying the failing set conditions) can be skipped without further computation. This technique is later adopted by RM [39] and VEQ [22].

GuP [3] extends the failing set idea by introducing guard-based pruning, which retains discovered unpromising partial matches for repeated pruning at the cost of additional memory usage. However, these methods focus solely on pruning invalid extensions and do not reuse valid historical extension results.

VEQ [22] proposes a pruning strategy based on dynamic equivalence, which can eliminate equivalent search subtrees, including both promising and unpromising ones. For each query vertex u, a dynamic equivalence class is defined over the filtered candidate set $C(u)$. Two candidates v_i and v_j in $C(u)$ belong to the same class if they share common neighbors, i.e., $\mathcal{A}^u_{u'}(v_i) = \mathcal{A}^u_{u'}(v_j)$ for every $u' \in N(u)$. This idea was later also adopted by BICE [8]. However, its effectiveness depends on the number of equivalent candidate vertices. As a result, VEQ may perform poorly on data graphs with high average degrees or limited equivalence among candidates.

Forward-Looking Optimization. Forward-looking optimization does not utilize historical results. Instead, it directly merges certain extension branches when it predicts that they will produce similar extension results.

Circinus [20] proposes a merging strategy based on a vertex cover of the query graph. For a query vertex u_i not included in the vertex cover, all of its backward neighbors must be in the cover. In such cases, given a partial embedding M, Circinus merges the candidate mappings of u_i and compresses multiple extended partial embeddings into a group, delaying their expansion until u_i's forward neighbors are processed. On the other hand, if u_i is in the vertex cover and has at least one backward neighbor not in the cover, then the compressed embeddings will be decomposed. Specifically, the merged mapping set of this backward neighbor will be split according to the different mappings of u_i, and each decomposed embedding will be handled separately. A continuous subgraph matching method, CaLiG [44], also adopts this strategy.

BSX [28] models the search space as a multi-dimensional search box. During the backtracking enumeration phase, it iteratively selects one dimension, which corresponds to a query vertex from the vertex cover, and chooses several similar data vertices for that dimension. Then, it refines the other dimensions accordingly. At the final enumeration layer, the homomorphic matches within the search box are verified and form the final matching results.

Some other works aim to identify unmatchable search branches in advance. For example, BICE [8] prunes branches that will result in conflicts by applying bipartite matching on the bipartite graph between unmapped query vertices and their candidate data vertices.

3 THEORETICAL FOUNDATION

In this paper, we propose Common Extension Merging (CEM) and Common Extension Reusing (CER) techniques, which can be viewed as *forward-looking* and *backward-looking* optimizations, respectively. Before diving into the details, we first study the following lemma, which lays the foundation for both CEM and CER. The proof is provided in the full version of this paper [43].

Lemma 1. *Given a query graph Q and a matching order $O = (u_0, u_1, \ldots, u_{n-1})$, let M_1 and M_2 be matches of the subquery Q_i induced by the first i vertices. Assume that M_1 and M_2 share the same mappings for all backward neighbors of u_i (i.e., $N^O_-(u_i)$), and let v be a data vertex that does not conflict with any vertex in M_1 or M_2. Then, if $M_1 \oplus v$ is a valid match of the subquery Q_{i+1}, so is $M_2 \oplus v$. Here, $M \oplus v$ denotes extending M along the matching order by appending the mapping $(u_{|M|}, v)$.*

Let us revisit Example 1. Two partial embeddings M_1 and M_2 for Q_4 share the same mappings for u_0 and u_1, which are the backward neighbors of u_4. Therefore, for each candidate vertex v of u_4 (i.e., $v \in \{v_7, v_8\}$), if $M_1 \oplus v$ forms a valid match of the subquery Q_5, then so does $M_2 \oplus v$.

Lemma 1 offers the most effective opportunity to eliminate redundant extensions during enumeration, as all valid extensions of u_i for M_1 can be directly reused for M_2 without any additional computation. In BFS-based approaches, when extending u_i, we can group all partial embeddings that share the same mappings for u_i's backward neighbors to avoid repeated work. However, doing so is non-trivial in DFS-based methods. Our proposed techniques, CEM (Section 4) and CER (Section 5), serve as practical relaxations of this lemma to facilitate extension sharing in DFS-based enumeration.

4 COMMON EXTENSION MERGING

The intuition behind CEM is to merge multiple partial embeddings into an aggregated embedding and extend them together. For instance, if u_i is not connected to u_{i+1}, all mapped data vertices of u_i can be aggregated together, since extending u_{i+1} does not depend on u_i (according to Lemma 1). Based on this, we propose the following *black-white vertex encoding* and *aggregated embedding*.

4.1 Black-white Vertex Encoding & Aggregated Embedding

Definition 4.1 (Black-White Vertex Encoding & Aggregated Embedding). Given a query graph Q with a matching order $O = (u_0, u_1, \dots, u_{n-1})$, each query vertex $u_i \in V(Q)$ is assigned a color $c(u_i) \in \{\text{black}, \text{white}\}$. An *aggregated embedding* M represents multiple embeddings and satisfies:

- $M[u]$ is a single data vertex if $c(u) = \text{black}$;
- $M[u]$ is a set of data vertices if $c(u) = \text{white}$.

If there is no ambiguity, the "embedding" mentioned in the rest of this paper refers to an aggregated embedding. And we denote the black (white) backward neighbors of u as $BK(u)$ ($WT(u)$).

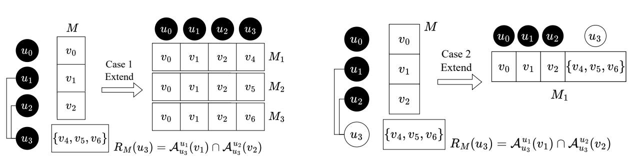
Example 3. *Consider again the query graph Q and data graph G in Figure 1. Let the matching order be $O = (u_0, \cdots, u_6)$, where u_3 and u_6 are encoded as white vertices. When $u_0, u_1,$ and u_2 are mapped to $v_0, v_1,$ and v_3, respectively, u_3 has only one candidate v_6, yielding the aggregated embedding $\{(u_0, v_0), (u_1, v_1), (u_2, v_3), (u_3, \{v_6\})\}$. In contrast, mapping (u_0, u_1, u_2) to (v_0, v_1, v_2) produces $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, \{v_4, v_5, v_6\})\}$, which compactly represents three embeddings.*

4.2 Four Cases of Extension

Based on the black-white vertex encoding, we propose a common extension merging enumeration framework, as shown in Algorithm 3. This framework computes the extensions of Q_{i+1} in the form of aggregated embeddings, based on an aggregated embedding M of Q_i. In summary, there are four cases for extending u_i, classified according to the encoding of u_i and the encodings of its backward neighbors $N^O_-(u_i)$.

Case 1. u_i is a black vertex, and all its backward neighbors $N^O_-(u_i)$ are black vertices.

Case 1 is equivalent to the standard extension framework described in Algorithm 2. In this case, we directly compute the set of extensible vertices $R_M(u_i)$, and extend M by mapping u_i to each data vertex $v \in R_M(u_i)$, resulting in an embedding $M \oplus v$ of Q_i. Figure 2a illustrates an example of Case 1. For illustration purposes, we treat u_3 as a black vertex in this example.



(a) Case 1. (b) Case 2.

Figure 2: Illustrations of Case 1 and Case 2.

Case 2. u_i is a white vertex, and all its backward neighbors $N^O_-(u_i)$ are black vertices.

White encoding of u_i indicates an opportunity for extension merging. Specifically, after computing the extensible vertices $R_M(u_i)$ (same as Case 1), we aggregate them as the mapping for u_i to form an aggregated embedding $M \oplus R_M(u_i)$. Figure 2b illustrates this extension pattern. Note that in Case 2, the extension branches $\{M \oplus v_i \mid v_i \in R_M(u_i)\}$ are merged into a single aggregated embedding to reduce redundant extensions.

Algorithm 3: Enumerate(Q, $\mathcal{A}$, O, $\mathcal{M}$, M, i) (black-white enumeration framework)

Input: The query Q, auxiliary structure $\mathcal{A}$, matching order O, result set $\mathcal{M}$, an (aggregated) embedding M of Q_i, and the backtracking depth i.

1 **if** $i = |V(Q)|$ **then**
2   Append valid full embeddings in M to $\mathcal{M}$; **return**
3 $BK(u_i)(WT(u_i)) \leftarrow \{u \mid c(u) = black\ (white), u \in N^O_-(u_i)\}$
4 **if** $WT(u_i) = \emptyset$ **then** // Case 1 or Case 2
5     $R_M(u_i) \leftarrow \cap_{u_j \in BK(u_i)} \mathcal{A}^{u_j}_{u_i}(M[u_j])$
6   **if** $c(u_i) = black$ **then** // Case 1
7   **foreach** $v \in R_M(u_i)$ **do**
8   Enumerate (Q, $\mathcal{A}$, O, $\mathcal{M}$, $M \oplus v$, $i+1$)
9   **else** // Case 2
10   Enumerate (Q, $\mathcal{A}$, O, $\mathcal{M}$, $M \oplus R_M(u_i)$, $i+1$)
11 **else** // Case 3 or Case 4
12   **if** $BK(u_i) \neq \emptyset$ **then**
13         $R_M(u_i) \leftarrow \cap_{u_j \in BK(u_i)} \mathcal{A}^{u_j}_{u_i}(M[u_j])$
14   **else**
15         $u_j \leftarrow \arg\min_{u \in WT(u_i)} |M[u]|$
16         $R_M(u_i) \leftarrow \cup_{v \in M[u_j]} \mathcal{A}^{u_j}_{u_i}(v)$
17   **if** $c(u_i) = black$ **then** // Case 3
18   **foreach** $v \in R_M(u_i)$ **do**
19             $M_v \leftarrow M$
20   **foreach** $u_j \in WT(u_i)$ **do**
21                 $M_v[u_j] \leftarrow M[u_j] \cap \mathcal{A}^{u_i}_{u_j}(v)$
22   **if** $\forall u_j \in WT(u_i), M_v[u_j] \neq \emptyset$ **then**
23   Enumerate (Q, $\mathcal{A}$, O, $\mathcal{M}$, $M_v \oplus v$, $i+1$)
24   **else** // Case 4
25         $\mathcal{S} \leftarrow$decompose $WT(u_i)$'s mappings
26   **if** $|\mathcal{S}| \geq |R_M(u_i)|$ **then** // Case 4.1
27   Same pseudocode as lines 18-23
28   **else** // Case 4.2
29   **foreach** $M_t \in \mathcal{S}$ **do**
30                 $R_{M_t}(u_i) \leftarrow \cap_{u_j \in N^O_-(u_i)} \mathcal{A}^{u_j}_{u_i}(M_t[u_j])$
31   Enumerate (Q,$\mathcal{A}$, O, $\mathcal{M}$, $M_t \oplus R_{M_t}(u_i)$, $i+1$)

Case 3. u_i is a black vertex, and at least one of its backward neighbors is a white vertex.

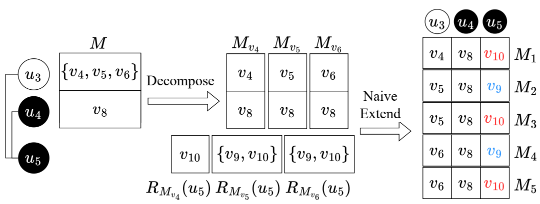
As illustrated in Figure 3a, M is an aggregated embedding of Q_5 in Figure 1, where the matching part $\{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$ is omitted for simplicity. The white vertex u_3 is mapped to three data vertices: $\{v_4, v_5, v_6\}$. When extending u_5, a naive solution may split M into three separate embeddings for its white backward neighbor u_3 and perform the extension as in Case 1. However, this leads to redundant computations. By preserving aggregation instead, we can merge the resulting embeddings M_1, M_3, M_5 into a single M', thereby facilitating more sharing opportunities for future extensions such as u_6.

To address this efficiently, we propose the solution illustrated in Figure 3b. We first compute the set of extensible vertices $R_M(u_i)$, which will be detailed later. Then, for each vertex $v_i \in R_M(u_i)$, we prune the candidate vertices v_j in the aggregated mapping $M[u_j]$ of each white backward neighbor u_j of u_i (lines 20-21). Specifically, if v_i is not adjacent to v_j, v_j is filtered out.

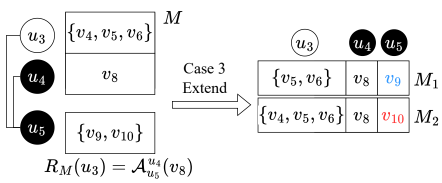
If the mapping set of any white backward neighbor becomes empty after pruning, v_i is removed from $R_M(u_i)$. Otherwise, v_i is appended to the shrunk aggregated embedding to form a valid extension (lines 22-23).

We now return to computing the extensible vertex set $R_M(u_i)$, which depends on whether u_i has black backward neighbors:

- **If u_i has at least one black backward neighbor** (lines 12-13), the extensible vertices are obtained by intersecting the adjacency sets of all such neighbors: $R_M(u_i) = \bigcap_{u_j \in BK(u_i)} \mathcal{A}_{u_i}^{u_j}(M[u_j])$.
- **If u_i has no black backward neighbors** (lines 14-16), we select the white neighbor $u_j \in WT(u_i)$ with the minimum $|M[u_j]|$ and compute the extensible vertices by taking the union of adjacency sets: $R_M(u_i) = \bigcup_{v_j \in M[u_j]} \mathcal{A}_{u_i}^{u_j}(v_j)$.



(a) Naive solution Case 3



(b) Our solution for Case 3

Figure 3: Illustrations of Case 3.

Case 4. u_i is a white vertex, and at least one of its backward neighbors is a white vertex.

We first compute the extensible vertices $R_M(u_i)$ in the same manner as in Case 3. The key challenge in this case lies in how to handle the aggregated embeddings involving u_i and its white backward neighbors. Since both u_i and some of its backward neighbors are white, we consider two alternative strategies.

One is to retain the aggregated embeddings of u_i's white backward neighbors and treat u_i as a black vertex, i.e., process each mapping in $R_M(u_i)$ individually. The other is to break the existing aggregated embeddings into finer-grained ones, and merge u_i's mappings into the aggregation to form new composite embeddings.

In our design, the choice between these two strategies is adaptive depending on which one leads to less redundant computation. Specifically, we divide Case 4 into two subcases:

Case 4.1 (lines 26-27). If the size of the Cartesian product of the mapping sets of u_i's white backward neighbors is not less than $|R_M(u_i)|$, it is more efficient to process u_i separately. Thus, we retain the aggregated embeddings of its white backward neighbors and extend each $v_i \in R_M(u_i)$ independently. This is essentially the same strategy as in Case 3.

Case 4.2 (lines 28-31). If the size of the Cartesian product of the mapping sets of u_i's white backward neighbors is smaller than $|R_M(u_i)|$, merging the mappings of u_i into the aggregation may reduce redundancy. In this case, we enumerate all embeddings in the Cartesian product set $\mathcal{S}$, and for each embedding $M_t \in \mathcal{S}$, we extend it with u_i following the procedure used in Case 2.

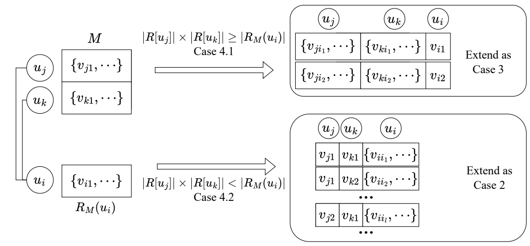


Figure 4: Illustration of Case 4.

Figure 4 illustrates an example of this extension strategy. Suppose the white vertex u_i has two white backward neighbors, u_j and u_k. If the product of their candidate set sizes satisfies $|R[u_j]| \times |R[u_k]| \geq |R_M(u_i)|$, we apply the same strategy as in Case 3 (upper part of Figure 4) to directly extend u_i. Otherwise, we decompose the candidate sets of $R[u_j]$ and $R[u_k]$, and keep u_i's candidates in an aggregated form, as in Case 2 (lower part of Figure 4).

Discussions. Compared with existing methods [20, 28, 44], our approach does not rely on a vertex cover of the query graph and can operate with any black-white encoding of the query graph, offering greater flexibility. In particular, these methods do not handle Case 4, which may miss opportunities for merging common extensions.

4.3 Dealing with Vertex Conflicts

One key challenge in subgraph isomorphism is enforcing the vertex injectivity constraint, which requires that each query vertex must be mapped to a unique data vertex. We refer to any violation of this constraint as a *vertex conflict*. Some existing methods, such as BSX [28], postpone conflict checking until the last level of enumeration. However, this may miss early pruning opportunities, as our experimental results demonstrate (Section 7.2.1).

Due to the existence of white vertices in our enumeration framework, our approach to detecting vertex conflicts differs slightly, and is not explicitly shown in Algorithm 3. In our solution, we only record the mappings of black vertices, as well as the mappings of white vertices when they are mapped to a single data vertex. This includes two situations: (i) u_i is deterministically mapped when extended under Case 3 or Case 4; and (ii) under Case 4, a white backward neighbor u_k of u_i becomes deterministically mapped when its candidate set is reduced to a single vertex. Such deterministically mapped vertices are included in conflict checking to preserve correctness and maximize pruning opportunities.

At the final level of enumeration, we obtain an aggregated embedding M in which every query vertex has been assigned a mapping. Since each white vertex may map to multiple candidate data vertices, we generate all possible full embeddings by taking the Cartesian product of their mappings. Each resulting embedding is then checked against the vertex injectivity constraint, and only valid ones are added to the result set $\mathcal{M}$ (lines 1-2 in Algorithm 3).

5 COMMON EXTENSION REUSING

`CEM` can share the extension computation in an aggregated embedding, but it still misses other optimization opportunities, as illustrated in the following example.

Example 4. *In Figure 5, we extend u_4 in the query graph given in Figure 1. Because u_4's backward neighbors are u_0 and u_1, Lemma 1* *tells us that the extension of u_4 only depends on u_0 and u_1, while u_2 and u_3 are* irrelevant. *Using* CEM*, assume that we have generated two partial embeddings M_1 and M_2 for subquery Q_3, which have the same mappings of u_0 and u_1, but different in u_2 and u_3. When extending u_4, we need to extend M_1 and M_2, respectively. However, the extension of u_4 is the same for both M_1 and M_2. Thus, a desirable solution should avoid such duplicated extension, which cannot be achieved by* CEM.

To address the above problem, we propose a *backward-looking* common extension reusing technique, called CER, which uses the historical results already generated to avoid duplicated extension.

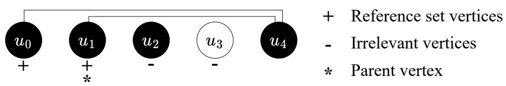


Figure 5: The reference set and parent vertex of u_4

5.1 Reference Set & Brother Embeddings

We first give the definitions of *reference set* and *brother embeddings*, which are main concepts to introduce CER. The idea behind them is based on a relaxed version of Lemma 1.

Definition 5.1 (Reference Set). Assume that the next query vertex to be extended is u_i. The *reference set* of u_i is a subset of $\{u_0, \cdots, u_{i-1}\}$, denoted as $RS(u_i)$, includes two components:

- The closure of u_i's backward neighbors, denoted as $Anc(u_i)$, which contains $N_-^O(u_i)$ and all their backward neighbors recursively;
- All vertices u_k (with $k < i$) that are adjacent to at least one white backward neighbor of u_i.

Formally, the reference set $RS(u_i)$ is defined as:

$$RS(u_i) = Anc(u_i) \cup \{u_k \mid \exists u_j \in WT(u_i),\ e(u_j, u_k) \in E(Q),\ k < i\}. \quad (1)$$

Intuitively, the reference set $RS(u_i)$ contains the query vertices whose mappings affect the extensible set $R_M(u_i)$. Besides the direct backward neighbors of u_i, it also includes vertices u_k ($k < i$) that are adjacent to any white backward neighbor $u_j \in WT(u_i)$. This is because u_k is extended using Case 3 or Case 4, which could prune the mappings of u_j, and thereby influence the extension of u_i.

Definition 5.2 (Brother Embeddings). Assume that M_1 and M_2 are two partial embeddings of the subquery Q_i, and the next query vertex to be extended is u_i. We say that M_1 and M_2 are *brother embeddings* if they assign the same mapping to every vertex in the reference set of u_i, i.e., $M_1(u) = M_2(u)$ for all $u \in RS(u_i)$.

Example 5. *Recall Example 1, M_1 and M_2 are two partial embeddings of the subquery Q_4. As shown in Figure 5, the next query vertex u_4 is adjacent to u_0 and u_1, both of which are black vertices. Therefore, the reference set of u_4 is $RS(u_4) = \{u_0, u_1\}$. In this case, M_1 and M_2 assign the same mappings to the vertices in $RS(u_4)$, i.e., $M_1[u_0] = M_2[u_0] = v_0$ and $M_1[u_1] = M_2[u_1] = v_1$. Thus, M_1 and M_2 are brother embeddings.*

5.2 CER Strategy for Extension Reusing

We now propose the CER strategy. Given a query graph Q, for each query vertex u_i, we define its *parent vertex* as the vertex $u_k \in RS(u_i)$ with the largest index in the matching order O (e.g., in Figure 5, u_1 is the parent vertex of u_4). Correspondingly, u_i is referred to as a *child vertex* of u_k.

If u_k is not the immediate predecessor of u_i in O, i.e., $k < i - 1$, we set a flag $u_i.f$ = `true`, record its parent as $u_i.p = u_k$, and register u_i as a child of u_k via $u_k.child \leftarrow u_k.child \cup \{u_i\}$. Otherwise, we set $u_i.f$ = `false`. The CER strategy is applied only to vertices u_i with $u_i.f$ = `true`. We design a buffer structure, called the *Common Extension Buffer* (CEB), to cache reusable extension results.

Definition 5.3 (Common Extension Buffer (CEB)). Given a query vertex u_i with $u_i.f$ = `true`, its CEB consists of two components: $CEB(u_i).g$, a boolean flag indicating whether the buffer is valid (initialized to `false`); and $CEB(u_i).b$, which stores reusable local extensions of u_i.

We now describe how to use the CEB during the CER procedure. Suppose we are processing a partial embedding M and the next query vertex to be extended is u_i, where $u_i.f$ = `true`.

(1) When $CEB(u_i).g$ is `false`: This indicates the *first time* u_i is extended among M's brother embeddings. We update $CEB(u_i).b$ with the current extensions and set $CEB(u_i).g$ = `true` after processing. The contents pushed into the buffer depend on the extension strategy, as discussed in Section 4:
  - Case 1 & 2: Store $R_M(u_i)$ directly into $CEB(u_i).b$.
  - Case 3: Store $\{M_v \oplus v \mid v \in R_M(u_i)\}$, where M_v is the shrunk embedding derived from M associated with v (see lines 19-21 in Algorithm 3).
  - Case 4: For Case 4.1, store $\{M_v \oplus v \mid v \in R_M(u_i)\}$ as in Case 3; For Case 4.2, store $\{M_t \oplus R_{M_t}(u_i) \mid M_t \in \mathcal{S}\}$, where $\mathcal{S}$ is the Cartesian product of decomposed candidates, and $R_{M_t}(u_i)$ is the extensible set of u_i under M_t.

(2) When $CEB(u_i).g$ is `true`: This means that valid extensions have already been cached. We reuse $CEB(u_i).b$ to directly extend M based on u_i's different extension cases, avoiding redundant computation.

(3) When backtracking from the level of u_i in the search tree, we reset the CEB flags of all child vertices of u_i, i.e., we set $CEB(u_j).g$ = `false` for every $u_j \in u_i.child$.

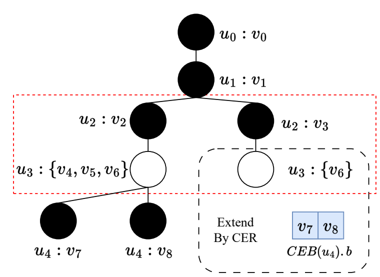


Figure 6: Using CEBs to eliminate duplicated computation

Example 6. *Referring to the query and data graphs in Figure 1, the search tree from u_0 to u_4 rooted at v_0 is illustrated in Figure 6. Assume that the enumeration has just backtracked to the partial embedding $M = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, \{v_4, v_5, v_6\})\}$ after the mappings $\{v_7, v_8\}$ for u_4 are cached in $CEB(u_4).b$. Since u_1 is the* *parent of u_4, all partial embeddings within the red dashed box represent the brother embeddings of M. Consequently, the embedding $\{(u_0, v_0), (u_1, v_1), (u_2, v_3), (u_3, \{v_6\})\}$ can be extended directly by utilizing the $CEB(u_4).b$. This CEB of u_4 remains valid until the mapping of u_1 changes from v_1.*

Remark. We enable CER only for query vertices u_i whose parent vertex is not u_{i-1}. The reason is that if u_i's parent is exactly u_{i-1}, then each time we backtrack to the depth of u_i, the mappings of vertices in $RS(u_i)$ may have changed due to the rematching of u_{i-1}. As a result, the previous extensible vertices of u_i are no longer valid, making extension reuse ineffective.

6 OPTIMIZATIONS AND EXTENSIONS

This section presents several key optimizations in CEMR, including two effective pruning techniques, matching order selection, and an encoding strategy. We further describe how CEMR can be extended to support directed and edge-labeled graphs.

6.1 Pruning Techniques

To effectively identify unpromising search branches that cannot produce any valid full embedding during enumeration, we additionally propose two pruning techniques.

6.1.1 *Contained Vertex Pruning.*

Definition 6.1 (Contained Vertex Set). Suppose the matching order is $O = (u_0, \cdots, u_{n-1})$, and u_i and u_j is two vertices with the same label, and $i < j$. If the backward neighbor set of u_i is u_j's subset, i.e., $N_{-}^{O}(u_i) \subseteq N_{-}^{O}(u_j)$, then we say that u_j is contained by u_i under the matching order O. The set of query vertices that contained by u_i under the matching order is called *contained vertex set* of u_i, denoted by $Con(u_i)$.

Then we have the following pruning rule, whose proof can be found in the appendix of the full version of this paper [43].

Lemma 2 (Contained Vertex Pruning). *During the enumeration process of extending u_i under the partial embedding M, if $|R_M(u_i)| < |Con(u_i)|$, then the search branch rooted at M can be safely pruned.*

6.1.2 *Extended Failing Set Pruning.*

The failing set is an effective backjumping-based pruning technique, originally proposed in DAF [16], and has been adopted by subsequent methods [8, 22, 39].

Definition 6.2 (Failing Set). Let M be a partial embedding. A subset of query vertices F_M is called a *failing set* of M if no valid full embedding can be extended from the restriction of M to F_M, denoted as $M[F_M]$.

Lemma 3 (Failing Set Pruning). *Suppose M is a partial embedding where the last matching is $(u_i, M[u_i])$, and F_M is a non-empty failing set of M. If $u_i \notin F_M$, then all sibling branches of M in the search tree can be safely pruned.*

Intuitively, the failing set F_M indicates those query vertices whose current mappings prevent the enumeration from producing any valid embeddings. If none of the mappings in F_M are changed, subsequent search will inevitably fail. Building on this idea, we propose an *extended failing set pruning* technique under the black-white vertex enumeration framework. Specifically, the failing set of a partial embedding M (where u_i is the last mapped vertex in M) is computed as follows:

(1) If M is a leaf node in the search tree.
- M belongs to *complete embedding class* if M contains valid matchings. Then $F_M = \emptyset$.
- M belongs to *insufficient candidate class* if $|R_M(u_i)| < Con(u_i)$. Then $F_M = RS(u_i)$, corresponding to contained vertex pruning.
- M belongs to *vertex conflict class* if the mapping of u_i conflicts with that of some u_j where $j < i$. Then $F_M = Con(u_i) \cup Con(Tr(u_j))$, where $Tr(u_j)$ denotes the query vertex that restricts the candidate set of u_j to a singleton. Specifically, if u_j is a black vertex, or a white vertex with $|R_M(u_j)| = 1$, then we set $Tr(u_j) = u_j$. Otherwise, if u_j is a white backward neighbor of some u_k (where $k < i$), and u_k is the first vertex in the matching order that reduces the mapping set of u_j to a single vertex, then we set $Tr(u_j) = u_k$.

(2) If M is an internal node in the search tree, and $M_1, \ldots, M_k$ are all its children.
- First, if there exists a child M_l such that $F_{M_l} = \emptyset$ (i.e., M_l leads to a valid embedding), then $F_M = \emptyset$.
- Else if there exists some M_l such that $u_i \notin F_{M_l}$, then $F_M = F_{M_l}$.
- Otherwise, $F_M = \bigcup_{l=1}^{k} F_{M_l}$.

Remark. Compared with the original failing set proposed in DAF [16], our solution introduces two main extensions. First, we incorporate the contained vertex pruning rule, which enables earlier detection of failure. Second, our approach is compatible with the black-white vertex enumeration framework, where a batch of data vertices may be mapped to a single white query vertex.

6.2 Matching Order Selection

We decouple the matching order from the encoding strategy to reduce complexity. Our goal in matching order selection is to minimize the size of intermediate results during backtracking, thereby reducing the search space.

After the filtering phase, each query vertex u is associated with a candidate set $C(u)$. We prioritize vertices with smaller candidate sizes and stronger connectivity to the current partial order. Specifically, the first vertex in the matching order is selected as:

$$u_0 = \arg\min_{u \in V(Q)} \frac{|C(u)|}{d(u)}. \tag{2}$$

Given a partial matching order $O_i = (u_0, u_1, \cdots, u_{i-1})$, denote $N(O_i) = \cup_{u \in O_i} N(u)$, and the next vertex is chosen by:

$$u_i = \arg\min_{u \in N(O_i) \setminus O_i} \frac{|C(u)|}{|N(u) \cap O_i|}. \tag{3}$$

6.3 Encoding Strategy

Given a matching order O, CEMR searches for embeddings by considering the encoding types (black or white) of the current query vertex u_i and its backward neighbors $N_{-}^{O}(u_i)$. These encoding choices influence both pruning strategies and the degree to which intermediate computations can be shared. An effective encoding for a query vertex u should balance the following factors:

- A large number of forward neighbors or forward-neighbor candidates increases the case 3 and 4 situations, resulting in redundant splitting costs.
- Encoding u as white is less beneficial if it has many white backward neighbors, since fewer constraints are enforced. In contrast, more black backward neighbors help confirm the matching of u, but reduce the possibility of computation sharing.
- Vertices with common labels are likely to cause conflicts during backtracking, and assigning them as white vertices would undermine the algorithm's ability to find such vertex conflict during intermediate search.
- A large candidate size for u suggests more potential for computation reuse among its candidates.

To jointly consider these factors, we propose a cost model that compares the white risk $WR(u)$ and the black risk $BR(u)$:

$$WR(u) = (1 + \sum_{u' \in N_{+}^{O}(u)} |C(u')|) \times |V(Q, L(u))| \times |WT(u)|; \quad (4)$$

$$BR(u) = |C(u)| \times |BK(u)|, \quad (5)$$

where $|V(Q, L(u))|$ is the number of query vertices of label u. We encode u as white if $WR(u) < BR(u)$, and as black otherwise.

6.4 Extensions

Although CEMR is described under the undirected, vertex-labeled graph model, it can be readily extended to directed and edge-labeled graphs. Only two components require modification:

- **Filtering stage.** Candidate edges must respect both the direction and the label of query edges during the filtering process.
- **Definition of containing vertex.** During enumeration, edge directions must be considered for containing vertices. For two vertices u_i and u_j ($i < j$), if both the in- and out-backward neighbors of u_i are subsets of those of u_j, then u_j is said to be contained by u_i, enabling containing pruning.

7 EXPERIMENTS

7.1 Experimental Setup

7.1.1 Datasets. We conduct our experiments on eight real-world datasets: Yeast, Human, HPRD, WordNet, DBLP, EU2005, YouTube, and Patents. These datasets are widely used in previous studies [3, 8, 16, 22, 39] and surveys [37, 47], and they span a variety of domains, including biology (Yeast, Human, HPRD), lexical semantics (WordNet), social networks (DBLP, YouTube), the web (EU2005), and citation networks (Patents). They differ in scale and complexity (e.g., topology and density), and their detailed statistics are summarized in Table 2.

7.1.2 Queries. We use 10,000 queries for each dataset in our experiments, covering a wide range of sizes and densities. Following prior work [3, 16, 37, 47], we generate query graphs by performing random walks on the data graphs and extracting the induced subgraphs. This procedure guarantees that every query graph has at least one valid embedding. For Human and WordNet (challenging due to topology denseness and label sparseness, respectively), we use query sizes of $\{4, 8, 12, 16, 20\}$, and for the other six datasets, we use $\{4, 8, 16, 24, 32\}$. For each query size n, we denote the corresponding query set as Q_n, which contains 2,000 queries. Thus, each dataset comprises a total of $2000 \times 5 = 10000$ query graphs. When $n > 4$, Q_n is further divided into a sparse subset Q_{nS} (average degree < 3) and a dense subset Q_{nD} (otherwise), i.e., $Q_n = Q_{nS} \cup Q_{nD}$, with 1,000 queries in each subset.

Table 2: Datasets statistics

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$	average degree
Yeast	3,112	12,519	71	8.0
Human	4,674	86,282	44	36.9
HPRD	9,460	34,998	307	7.4
WordNet	76,853	120,399	5	3.1
DBLP	317,080	1,049,866	15	6.6
EU2005	862,664	16,138,468	40	37.4
YouTube	1,134,890	2,987,624	25	5.3
Patents	3,774,768	16,518,947	20	8.8

7.1.3 Compared Methods. For performance comparison, we evaluate our CEMR algorithm with six state-of-the-art subgraph matching algorithms: DAF [16], RM [39], VEQ [22], GuP [3], BICE [8] and BSX [28]. All the source codes of the compared algorithms come from the original authors.

7.1.4 Experiment Environment. Except for GuP, which is implemented in Rust, all other algorithms are implemented in C++. We compile all C++ source code using g++ 13.1.0 with the -O2 flag. All experiments are conducted on an Ubuntu Linux server with an Intel Xeon Gold 6326 2.90 GHz CPU and 256 GB of RAM. Each experiment is repeated three times, and the mean values are reported.

7.1.5 Metrics. To evaluate performance, we measure the elapsed time in milliseconds (ms) for processing each query, which consists of two components: preprocessing time and enumeration time. Specifically, the preprocessing time includes filtering and ordering steps (lines 1-2 of Algorithm 1), and for CEMR, it also includes the encoding step. The enumeration time refers to the duration of the backtracking search (line 4 of Algorithm 1). Except for the experiment in Section 7.2.4, we terminate the algorithm after finding the first 10^6 results as default. To ensure queries complete within a reasonable time, we impose a timeout limit of 6 minutes; if a query exceeds this limit, its elapsed time is recorded as 6 minutes.

7.2 Comparison with Existing Methods

7.2.1 Total Time Comparison. Figure 7 presents the total query processing time of all compared methods. For clarity, we separate the enumeration time (bottom bars with hatching) from the preprocessing time (top solid bars). Except for HPRD, CEMR consistently outperforms all other methods, achieving a speedup of 1.39× to 9.80× over the second fastest method. This improvement is mainly achieved in the enumeration stage, owing to the extension-reduction techniques of CEMR, and it becomes more pronounced for queries with large result sizes (see Section 7.2.4).

HPRD is a relatively simple dataset with a large number of labels. In such cases, most of the runtime is dominated by filtering, and DAF performs filtering more efficiently on HPRD, resulting in a shorter total time than CEMR. However, as shown in Section 7.2.2, CEMR still surpasses DAF in enumeration on HPRD.

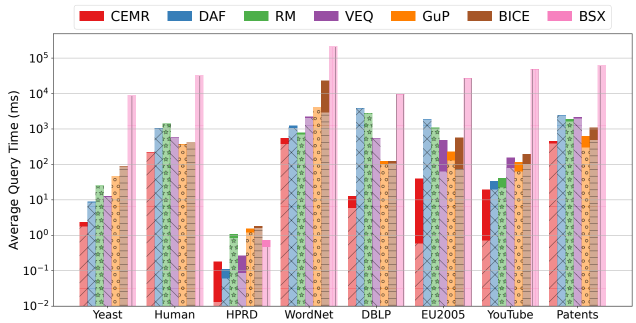


Figure 7: Average query time comparison. The bottom bars with hatching represent enumeration time, while the solid bars above represent preprocessing times.

Some methods involve additional preprocessing steps. For example, BICE clusters vertices that share the same neighbors after filtering, and GuP calculates guards for all candidates. These extra steps can lead to poor total performance on certain datasets (e.g., BICE on WorNet, GuP on Patents).

CEMR also outperforms BSX in terms of efficiency in our experiments. We attribute this performance gap to two main factors. First, CEMR incorporates highly effective pruning strategies, such as the failing set pruning, which are absent in BSX. Second, BSX only guarantees homomorphism during the batch search and postpones the injectivity check to the final enumeration stage. Consequently, it incurs unnecessary overhead by processing many intermediate candidates that appear feasible but ultimately fail to satisfy the injectivity constraint.

7.2.2 Enumeration Time Comparison. Since CEMR is primarily designed to accelerate the enumeration phase, we conduct a detailed evaluation of its enumeration performance. From the results in Figure 7, CEMR processes queries faster than other methods in most cases (with the exception of Patents, due to one additional timeout query, as discussed in Section 7.2.3). Compared with the second-best method, CEMR achieves a speedup ranging from 1.67x to 108.52x. As will be shown in Section 7.3, this advantage largely stems from the proposed CEM and CER techniques.

We further analyze enumeration time under different query sizes. The results in Figure 8a show a consistent trend across datasets: enumeration time increases with query size for all methods, and CEMR generally outperforms its competitors. A notable case is EU2005, where the correlation between query size and enumeration time does not strictly hold. In this dataset, larger query graphs, after filtering, almost always yield matches with large result sets. Consequently, CEMR can quickly generate embeddings and enumerate a substantial number of results by redundant extension elimination, leading to comparable or even shorter enumeration times for larger queries compared with smaller ones.

To better understand the performance of CEMR, we compare the enumeration time of each query with that of the second-best method based on the average enumeration time on the largest-size queries in the corresponding dataset. The scatter plots in Figure 9 present the results for Yeast and YouTube, where the second-best methods are VEQ and DAF, respectively. In both cases, all queries are processed within the time limit by both methods. Most points lie above the diagonal line, indicating that CEMR achieves shorter enumeration times on the vast majority of queries. The improvement is particularly pronounced for queries with large result sets, where CEMR benefits from rapid embedding generation. For smaller queries, the advantage is smaller but still consistent, suggesting that CEMR not only scales well with query size but also maintains low per-query overhead. These results confirm that the performance gain of CEMR is not limited to a few extreme cases but is robust across different query characteristics.

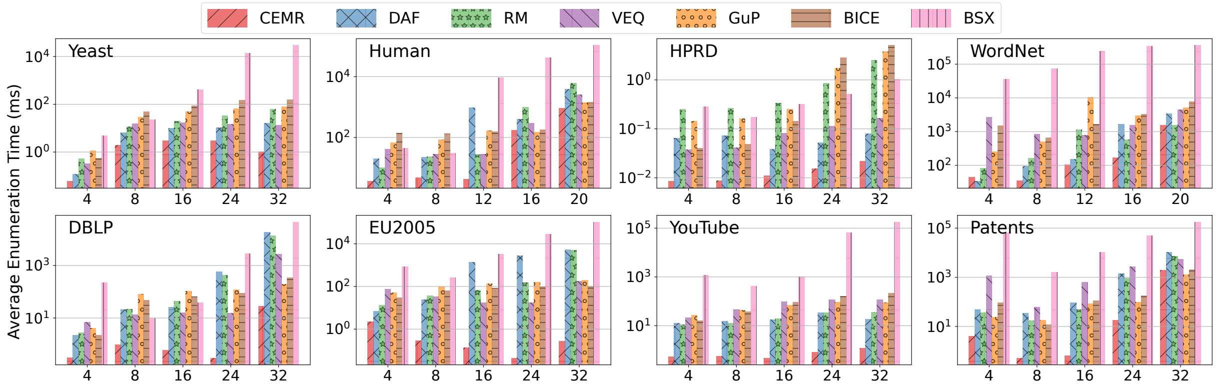
Table 3: Number of unsolved queries

Methods	Human	WordNet	DBLP	EU2005	Patents
DAF	25	16	86	50	56
RM	33	4	58	22	34
VEQ	14	9	13	**0**	22
GUP	7	48	**0**	**0**	**6**
BICE	**5**	16	**0**	**0**	10
BSX	774	5155	171	666	1042
CEMR	**5**	**0**	**0**	**0**	10

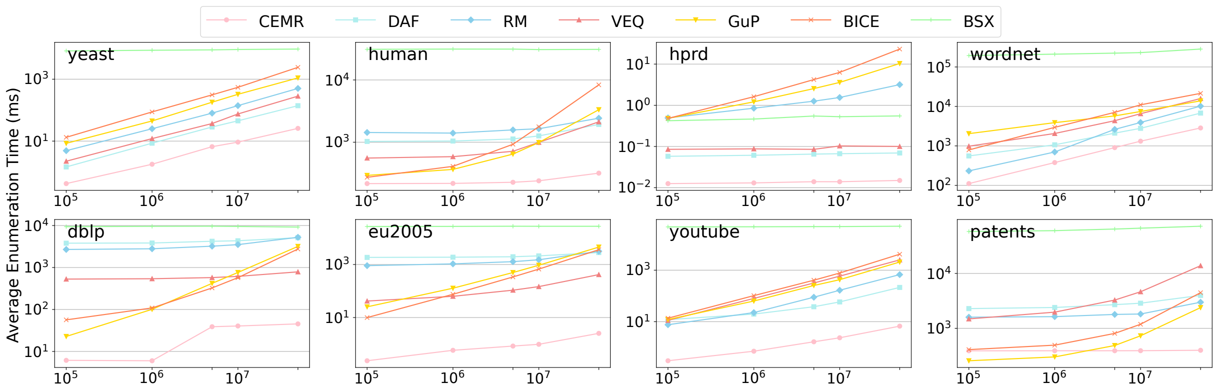
7.2.3 Unsolved Queries Number Comparison. The number of unsolved queries is an important metric for evaluating the performance of subgraph matching algorithms, as it reflects the algorithm's capability to handle difficult queries. Table 3 reports the number of unsolved queries across five datasets, excluding HPRD, Yeast, and YouTube, where all methods, except BSX, successfully return results within the time limit due to the simplicity of these datasets (BSX encounters 0, 179, and 1236 timeouts on HPRD, Yeast, and YouTube, respectively). A query is considered unsolved if it cannot be answered within 6 minutes. On most datasets, CEMR yields fewer unsolved queries than other methods, demonstrating its effectiveness in reducing redundant computation and in applying pruning strategies that prevent excessive exploration of unpromising branches.

7.2.4 Enumeration Time Comparison Varying the Limit Number. To test the embedding generation speed of different algorithms, we vary the limit number from 10^5 to $5{\times}10^7$ and record the enumeration times to generate 100K, 1M, 5M, 10M, and 50M embeddings for each algorithm. Note that we report the total enumeration time instead of the EPS (embeddings per second) metric, as the mean EPS is often dominated by extreme values (detailed EPS results are provided in the appendix of the full version of this paper [43].). Figure 8b shows the enumeration time comparison as the limit number varies. We observe that as the limit number increases, CEMR exhibits a lower runtime growth rate, mainly due to the black-white encoding technique, which allows CEMR to generate a batch of results at the same time. In contrast, GuP lacks a grouping technique, while DAF and RM only group matches of leaf vertices, causing their enumeration time to increase significantly with the limit number.

7.2.5 Memory Usage Comparison. Memory consumption is a critical factor in the practical deployment of subgraph matching algorithms. In this experiment, we compare the average peak memory consumption of CEMR with that of other methods. The results are presented in Table 4. To reduce the overhead of memory allocation during execution, we pre-allocate a fixed amount of memory.

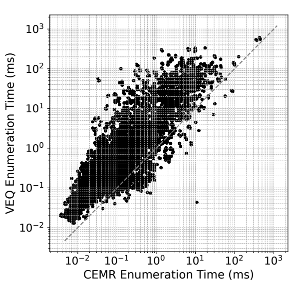


(a) Enumeration time comparison across different query sizes.

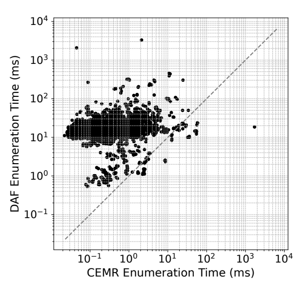


(b) Enumeration time comparison under different result size limits.

Figure 8: Detailed enumeration time comparison results. (a) Varying query sizes; (b) Varying result size limits.



(a) Yeast



(b) YouTube

Figure 9: Enumeration time comparison on Yeast and YouTube against VEQ and DAF, respectively.

Consequently, CEMR consumes more memory than other methods for small data graphs (e.g., Yeast, Human, and HPRD). However, as the data graph size increases, the memory usage of CEMR becomes comparable to DAF and RM, and is slightly lower in practice due to a more efficient implementation. Compared to VEQ and GuP, CEMR uses less memory because it does not require storing additional information in the auxiliary structure $\mathcal{A}$ for pruning.

7.3 Ablation Studies of CEMR

In this section, we conduct experiments to evaluate the effectiveness of our proposed techniques, including CEM, CER, pruning techniques, and matching orders.

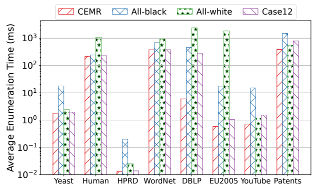
Table 4: Average Peak Memory Usage (MB)

Datasets	CEMR	DAF	RM	VEQ	GuP	BICE	BSX
Yeast	30.8	6.0	6.4	9.6	**5.2**	9.9	33.2
Human	36.3	**7.7**	9.6	15.2	12.2	15.4	18.2
HPRD	42.1	28.0	9.2	16.6	**8.3**	18.3	10.0
WordNet	116.2	**27.4**	62.8	125.8	130.1	151.8	1249.5
DBLP	92.2	**84.5**	110.9	261.1	112.4	248.1	178.0
EU2005	**536.6**	639.8	726.3	1194.6	881.8	1399.6	1287.0
YouTube	**297.5**	372.3	349.1	838.2	351.7	771.6	488.3
Patents	**1028.0**	1189.6	1336.7	3180.8	1553.8	3169.9	2082.4

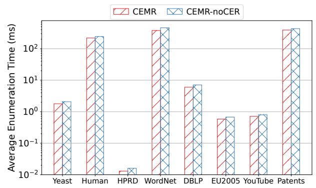
7.3.1 *Effectiveness of* CEM.

We design a black-white vertex encoding scheme for common extension merging (CEM in Section 4), and propose an encoding strategy guided by a cost model as described in Section 6.3. To investigate the effect of CEM, we compare the average enumeration time of CEMR with three variants, as shown in Figure 10a. In Figure 10a, All-black and All-white encode every query vertex as black and white, respectively, while Case12 encodes only the query vertices without forward neighbors as white, so there are no case 3 or case 4 extensions in Case12.

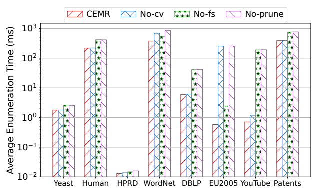
On easy datasets (e.g., Yeast, HPRD, YouTube, and Patents), most queries generate a large number of embeddings, and only a few search branches fail to yield valid results. In such scenarios, the aggregated mappings of white vertices help produce results more



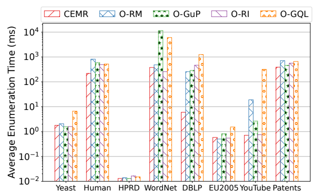
(a) Ablation study of CEM under different vertex encodings.



(b) Ablation study of CER.



(c) Ablation study of pruning techniques.



(d) Ablation study of matching order. O-* uses orders from other methods.

Figure 10: Ablation studies.

quickly, so both CEMR and All-white accelerate the enumeration process compared with All-black. On difficult datasets (e.g., Human, WordNet, DBLP, and EU2005), many redundant candidates arise from complex query topologies or vertex conflicts. Here, mapping certain query vertices to a single data vertex can better expose opportunities for pruning, and thus All-black performs better than All-white. We also observe that Case12 consistently outperforms All-black. Overall, since CEMR selects vertex encoding based on a cost model, it achieves the best performance among all four strategies.

7.3.2 Effectiveness of CER. CER uses common extension buffers to reuse extension results across different search branches. CEMR-noCER in Figure 10b disables the CER component of CEMR. The results in Figure 10b show that CER accelerates the enumeration process, achieving a 1.11x-1.23x speedup. This improvement comes from the reduced redundant computations and the efficient sharing of partial embeddings among search branches.

7.3.3 Effectiveness of Two Pruning Techniques. We conduct an ablation study on CEMR with three variants: No-cv, No-fs, and No-prune, which denote the variants without contained vertex pruning (Section 6.1.1), extended failing set pruning (Section 6.1.2), and both pruning techniques, respectively. As shown in Figure 10c, the full version of CEMR consistently outperforms all variants, indicating that both pruning techniques effectively reduce the search space and improve overall efficiency.

7.3.4 Effectiveness of Matching Order. To evaluate the matching order described in Section 6.2, we replace the original order in CEMR with those generated by RM [39] and GuP [3]. In addition, we include two well-established ordering methods recommended by a widely used survey [37], namely RI [6] and GQL [18]. We do not compare with DAF, VEQ, BICE, or BSX, since they adopt dynamic ordering strategies, where the next matched vertex may vary during execution. From the results shown in Figure 10d, we observe that the matching order produced by CEMR leads to more stable performance in most cases. Moreover, RM and GuP usually achieve faster enumeration time than those in Figure 7 (up to 203.9x). This indicates that these methods can also benefit from the redundant computation reduction techniques proposed in CEMR.

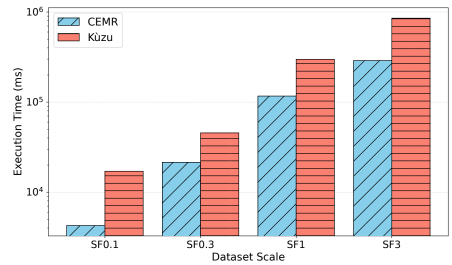


Figure 11: Execution time comparison on LSQB across different data scales.

7.4 Experiments on LSQB

To better demonstrate the effectiveness of our method, we conduct evaluations beyond comparisons with state-of-the-art subgraph query algorithms. Specifically, we compare CEMR with the high-performance open-source graph database Kùzu [12] on the LSQB benchmark [30]. LSQB is a subgraph query benchmark derived from LDBC SNB [10], which consists of directed graphs with both vertex and edge labels and focuses on complex multi-join queries typical in social network analysis. It contains 9 queries, and we modified the optional and negative clauses in q7, q8, and q9 into standard clauses, resulting in 9 connected query graphs. We did not set a running time threshold and terminated execution only when the method retrieved all matches.

As shown in Figure 11, which reports the total enumeration time aggregated over all nine LSQB queries, CEMR consistently outperforms Kùzu across all data scales, achieving speedups of 2.12x to 4.00x. These results demonstrate CEMR's efficacy on directed, multi-labeled complex workloads. The performance advantage stems from several factors: (1) our preprocessing-enumeration strategy uses effective filtering to prune unpromising search branches, reducing computational costs; and (2) as a lightweight prototype, CEMR avoids the overhead inherent in full-featured graph databases like Kùzu.

8 CONCLUSION

In this paper, we propose a subgraph matching algorithm CEMR for redundant extension elimination. We propose a forward-looking optimization CEM based on black-white vertex encoding and a backward-looking optimization CER based on common extension buffers. They effectively improve the performance of enumeration. The experimental results show the effectiveness of CEMR.

ACKNOWLEDGMENTS

This work was supported by New Generation Artificial Intelligence National Science and Technology Major Project under grant 2025ZD0123304 and NSFC under grant 62532001. This is a research achievement of National Engineering Research Center of New Electronic Publishing Technologies. The corresponding author of this work is Lei Zou (zoulei@pku.edu.cn).

REFERENCES

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

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worstcase optimal lowmemory dataflows. *arXiv preprint arXiv:1802.03760* (2018).

[3] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[4] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[6] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, Suppl 7 (2013), S13.

[7] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*. 1–12.

[8] Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. 2023. Bice: Exploring compact search space by using bipartite matching and cell-wide verification. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2186–2198.

[9] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[10] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[11] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory*. 8–21.

[12] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu graph database management system. In *The Conference on Innovative Data Systems Research*, Vol. 7. 25–35.

[13] Michael R Garey. 1979. A Guide to the Theory of NP-Completeness. *Computers and intractability* (1979).

[14] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1067–1082.

[15] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting reuse for GPU subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4231–4244.

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

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

[18] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 405–418.

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

[20] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[21] Alpár Jüttner and Péter Madarasi. 2018. VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (2018), 69–81.

[22] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*. 925–937.

[23] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *arXiv preprint arXiv:1506.01973* (2015).

[24] Wooyoung Kim, Min Li, Jianxin Wang, and Yi Pan. 2011. Biological network motif detection and evaluation. *BMC systems biology* 5, 3 (2011), 1–13.

[25] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.

[26] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. 2024. NewSP: A New Search Process for Continuous Subgraph Matching over Dynamic Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3324–3337.

[27] Lorenzo Livi and Antonello Rizzi. 2013. The graph matching problem. *Pattern Analysis and Applications* 16 (2013), 253–283.

[28] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. BSX: Subgraph Matching with Batch Backtracking Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[29] Azka Mahmood, Hina Farooq, and Javed Ferzund. 2017. Large Scale Graph Matching (LSGM): Techniques, Tools, Applications and Challenges. *International Journal of Advanced Computer Science and Applications* 8, 4 (2017), 494–499.

[30] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–11.

[31] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076* (2019).

[32] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.

[33] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[34] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[35] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 625–636.

[36] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[37] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[38] Shixuan Sun and Qiong Luo. 2020. Subgraph matching with effective matching order and indexing. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 491–505.

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

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

[41] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Călin Iorgulescu, Petr Koupy, Jinsoo Lee, et al. 2021. {aDFS}: An Almost {Depth-First-Search} Distributed {Graph-Querying} System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 209–224.

[42] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 335–346.

[43] Linglin Yang, Xunbin Su, Lei Zou, Xiangyang Gou, and Yinnian Lin. 2026. CEMR: An Effective Subgraph Matching Algorithm with Redundant Extension Elimination. arXiv:2603.08037 [cs.DB] https://arxiv.org/abs/2603.08037

[44] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[45] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data*. 2049–2062.

[46] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 192–203.

[47] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–29.

[48] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351.

[49] Gaoping Zhu, Xuemin Lin, Ke Zhu, Wenjie Zhang, and Jeffrey Xu Yu. 2012. TreeSpan: efficiently computing similarity all-matching. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 529–540.

[50] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. 2011. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment* 4, 8 (2011), 482–493.