Cao, Hongtai, Qihao Wang, Xiaodong Li, Matin Najafi, Kevin Chen–Chuan Chang, and Reynold Cheng. "Large Subgraph Matching: A Comprehensive and Efficient Approach for Heterogeneous Graphs." In 2024 IEEE 40th International Conference on Data Engineering (ICDE), pp. 2972-2985. IEEE, 2024.

repository: https://github.com/hongtaicao/cao2024large

2024 IEEE 40th International Conference on Data Engineering (ICDE)

# Large Subgraph Matching: A Comprehensive and Efficient Approach for Heterogeneous Graphs

Hongtai Cao*, Qihao Wang*, Xiaodong Li†, Matin Najafi†, Kevin Chen-Chuan Chang*, Reynold Cheng†,
* *University of Illinois at Urbana-Champaign*, Urbana-Champaign, USA
† *University of Hong Kong*, Hong Kong SAR
*{hongtai2, qwang65, kcchang}@illinois.edu, †{xdli, matin, ckcheng}@cs.hku.hk

***Abstract*—The subgraph matching problem is crucial in graph analysis, involving identifying all instances of a given pattern $P$ within a graph $G$. Advances in this field aim to uncover larger patterns across diverse graph types and subgraph matching tasks. However, existing methods often prove inefficient for such tasks. To address this gap, we propose CSCE, which generates efficient plans for various problem settings. CSCE utilizes clustered compressed sparse rows for heterogeneous graphs and sequential candidate equivalence to reduce redundant computations. Moreover, our approach seamlessly supports different subgraph matching variants, such as edge-induced, vertex-induced, and homomorphic scenarios. Experiments show that our work is up to two orders of magnitude faster than the state of the art on graphs of millions scale.**

***Index Terms*—subgraph matching, heterogeneous graphs, edge induce, vertex induce, homomorphism, clustering, equivalence**

## I. Introduction

Graphs, consisting of vertices and edges, are widely utilized to model real-world data in various domains. Subgraph Matching (SM), a fundamental task in graph analysis, finds instances of a *pattern graph* ($P$), within another graph, known as the *data graph* ($G$) [1]. For example, it efficiently builds a graph $G_P$ from $G$ to enable higher-order graph analysis [2]. Each edge $(v_i, v_j)$ of $G_P$ represents the frequency of subgraph instances of $P$ in $G$ containing vertices $v_i$ and $v_j$. Recent studies have found large subgraphs to be useful. In molecular networks, Spirin et al. [3] identify protein complexes and functional units as subgraphs with up to 35 vertices. In yeast proteome networks, Chen et al. [4] reveal functional modules comprising up to 98 vertices. DPCMNE [5] further detects protein complexes of up to 360 vertices. An exemplar complex from DPCMNE [5], illustrated in Fig. 1 as $P$ with 8 vertices, is arbitrarily labeled and oriented, serving as a recurring example.

Although useful in practice, SM is known to be NP-hard [6]. SM takes $P$ (pattern) and $G$ (data graph) as inputs, identifying isomorphic instances, called *embeddings*, of $P$ within $G$ as output. Existing algorithms determine a vertex sequence for $P$, known as the *matching order* or *plan* ($\Phi$), which is then executed to identify all embeddings. In Fig. 1, $P$ represents a *pattern*, and $G$ is a *data graph*. Vertices are distinguished as $u_i$ and $v_i$ for pattern and data graphs, respectively.

We focus on *heterogeneous* graphs. Unlike homogeneous graphs, heterogeneous graphs [7] accommodate different types of vertices and edges, capturing the multifaceted nature of data and thereby enhancing the depth and accuracy of analysis [8]. As $P$ is heterogeneous, vertices $u_1$, $u_2$ are labeled as $A$, $B$ respectively, and connected by a directed edge $u_1 \rightarrow u_2$.

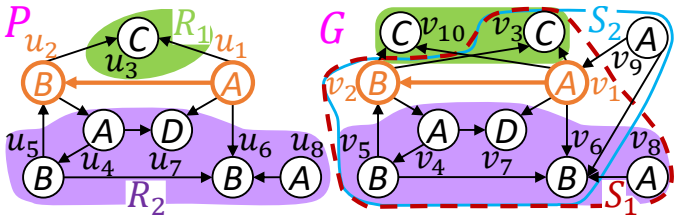


Fig. 1: A pattern $P$, a data graph $G$, and two subgraphs $S_1$ (red dashed circle) and $S_2$ (blue solid circle).

This paper targets large patterns, defined as those with 8 or more vertices following established literature [9]–[11], with a focus on common practices between 8 and 200 vertices *e.g.*, DAF [12] and VEQ [6]. Furthermore, as subgraphs are extracted differently in specific application scenarios, we study all SM variants, edge-induce, vertex-induce (higher-order graph analysis [2]), and homomorphism (graph databases [13]). Given the complexity of the problem, current state-of-the-art methods are inefficient as shown below.

**Heterogeneity-Aware Indexing:** Existing methods, *e.g.*, label degree filtering (LDF) and neighborhood label frequency filtering (NLF) [14], [15], exhibit repetitive label matching due to heterogeneity. In Fig. 1, existing methods find candidates for each vertex in $P$ by iteratively matching vertices in $G$ on labels and neighbors. For example, $v_1$ ($v_2$) can be a candidate for $u_1$ ($u_2$) because $v_1$ ($v_2$) has a neighbor $v_3$ with label $C$, and $v_3$ for $u_3$ due to matching labels. Note that we check the label of $v_3$ multiple times, one for each neighbor of $u_3$ and one for $u_3$ itself, leading to increased repetition as the pattern size grows. To eliminate such repetition, we propose heterogeneity-aware indexing to index vertices by their heterogeneous characteristics (labels and neighbors), so that we can look up candidates directly for every vertex of $P$. The indexing is realized as *Clustered Compressed Sparse Row* (CCSR), a novel data structure for $G$.

**Dependency-Aware Candidates:** Existing methods overlook a key concept, *dependency*. In Fig. 1, to match $P$ in $G$, existing solutions, *e.g.*, [11], [16], after matching the orange edges, $(u_1, u_2)$ with $(v_1, v_2)$, for each mapping, $v_3$ and $v_{10}$, of $u_3$ in the green region $R_1$, rematch the purple $R_2$ to

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00231

discover new instances of $P$. As the vertices in $R_1$ and $R_2$ are only connected by orange vertices, the mappings in $R_1$ and $R_2$ are conditionally independent given those of $u_1$ and $u_2$. Therefore, rematching is redundant. Such redundancy is severe in larger patterns as conditionally independent regions can be more frequent and larger. To avoid rematching in a matching sequence, *e.g.*, matching in $R_1$ and $R_2$ after the orange edges, our key insight, Sequential Candidate Equivalence (SCE), reveals vertices whose candidates are equivalent and reusable regardless of candidates of certain other vertices.

**Optimization:** Lastly, existing methods often have limited support for SM variants. For example, failing set pruning [6], [11], [12] and compression [15] only apply to edge-induced SM because they use its property of allowing arbitrary connection between data vertices as mappings of unconnected pattern vertices. Factorization [17] mainly applies to homomorphic SM [13] where Cartesian products are common. In contrast, we propose a variant-aware optimization algorithm on top of CCSR and SCE to support all variants. Our CCSR boosts performance on heterogeneous graphs. Meanwhile, our SCE extends all the above techniques as they use variant-specialized independence. In addition, as real-world graphs are sparse, we propose a Greatest-Constraint-First (GCF) heuristic that prioritizes matching edges for rapid candidate pruning. To maximize reusing candidates, we also propose a Largest-Descendant-Size-First (LDSF) heuristic.

**Method:** We propose CSCE (CCSR and SCE) to efficiently match large patterns in heterogeneous graphs. To build indices for retrieving candidates, CSCE first clusters edges in $G$ and stores them as CCSRs. For a given SM task, it then selectively retrieves clusters and optimizes plans in three key steps. In step one, to quickly prune candidates in the matching process, it generates an initial plan by GCF. In step two, to enable reusing candidates, it identifies SCE by constructing a directed acyclic graph. In step three, to maximize reusing candidates, it further fine-tunes the plan by LDSF. Finally, CSCE is implemented to efficiently address all edge-induced, homomorphic, and vertex-induced SM variants.

To summarize, our contributions are:

- Motivating the significance of SM on large patterns, highlighting existing limitations in current approaches.
- Proposing a novel CCSR data structure and the concept of SCE to effectively address large pattern challenges.
- Developing GCF and LDSF heuristics to generate efficient matching plans.
- Enabling support for edge-induced, homomorphic, and vertex-induced SM on heterogeneous graphs, a capability not provided by existing works.
- Conducting extensive experiments that demonstrate our approach outperforms state-of-the-art methods by up to two orders of magnitude.

**Paper Organization:** Fig. 2 summarizes the paper workflow. Section II introduces preliminaries and related works. Section III presents our method overview. Sections IV, V, and VI propose CCSR, SCE, and plan optimization. Section VII presents experiment results and Section VIII concludes.

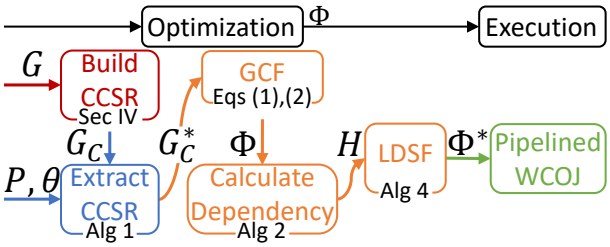


Fig. 2: Method overview.

## II. Preliminary and Related Work

In this paper, we focus on both undirected and directed graphs where vertices and edges can have labels [18]. Vertices [19], [20] are also called nodes [21]. A graph $G = (V_G, E_G, L_G, \Sigma_G)$ is a set of vertices $V_G$ and edges $E_G$ connecting vertices, where $L_G$ and $\Sigma_G$ are label functions on vertices and edges. Let $l_v$ and $l_e$ be numbers of labels on vertices and edges respectively. If $l_v + l_e > 2$ [7], then $G$ is *heterogeneous*; otherwise, it is *homogeneous*. To distinguish edge directions, we represent an edge $e_{ab} \in E_G$ as an ordered vertex pair $(v_a, v_b)$ [16] if $v_a \to v_b$, $v_a, v_b \in V_G$, where $v_a$ ($e_{ab}$) is called the incoming neighbor vertex (edge) of $v_b$ and $v_b$ ($e_{ab}$) is called the outgoing neighbor vertex (edge) of $v_a$ [1]. The degree $d(v)$ of a vertex $v$ is the number of its neighbor vertices. An undirected edge $v_a - v_b$ is represented as two directed edges $(v_a, v_b)$ and $(v_b, v_a)$. A graph is undirected if all edges are undirected, and is directed otherwise. As all vertices (edges) sharing the same label can be viewed as unlabeled, our graph definition directly supports unlabeled graphs. Note that we require $G$ to have no self-loops.

A subgraph $S$ of $G$ is a graph that contains a subset $V_S$ of $V_G$ and a subset $E_S$ of $E_G$, denoted as $S = (V_S, E_S, L_G, \Sigma_G)$. If a subgraph $S_v$ is induced from $V_{S_v}$, then $E_{S_v} = (V_{S_v} \times V_{S_v}) \cap E_G$ and $S_v$ is a vertex-induced [22] (induced [6], [12]) subgraph, denoted as $S_v = G[V_{S_v}]$ [12]. If a subgraph $S_e$ is induced from $E_{S_e}$, then $V_{S_e} = \cup_{(v_a,v_b)\in E_{S_e}} \{v_a, v_b\}$ and $S_e$ is an edge-induced [23] (non-induced [6]) subgraph. In comparison, given $V_S \subset V_G$, a vertex-induced subgraph contains all edges of $E_G$ between $V_S$ but an edge-induced subgraph contains a subset of $E_G$ between $V_S$. Therefore, a vertex-induced subgraph can be viewed as a special case of an edge-induced subgraph [22].

We next introduce graph isomorphism. Given two graphs $P$ and $S$, if there exists a bijective mapping, called *isomorphism*, $f : V_P \mapsto V_S$ such that $(u_i, u_j) \in E_P$ if and only if $(f(u_i), f(u_j)) \in E_S$, $L_P(u_k) = L_S(f(u_k))$ for $k = i, j$, and $\Sigma_P(u_i, u_j) = \Sigma_S(f(u_i), f(u_j))$, then $P$ and $S$ are isomorphic, denoted as $P \cong S$ [24]. As a remark, $P \cong P$ and is called *automorphism* [25], *e.g.*, a vertex-induced subgraph $S_3$ from $\{u_1, u_6, u_8\}$ is automorphic to itself under two mappings $f_1(u_i) = u_i$, $i = 1, 6, 8$ and $f_2(u_i) = u_{9-i}$, $i = 1, 8$, $f_2(u_6) = u_6$. In addition, if $f$ is surjective and $(u_i, u_j) \in E_P$ implies $(f(u_i), f(u_j)) \in E_S$, then $f$ is called *homomorphism* [11] and $P$ is homomorphic to $S$, *e.g.*, $S_3$ is homomorphic to an edge $(u_1, u_6)$ with $f_3(u_i) = u_1$, $i = 1, 8$, $f_3(u_6) = u_6$. In

other words, homomorphism allows mapping different pattern vertices to the same data graph vertex.

**Problem Statement.** Using the above concepts, we formally define our problem. Given two graphs $G$ and $P$, an *edge-induced* (a *vertex-induced*) SM problem finds all edge-induced (vertex-induced) subgraphs $S \subset G$ such that $P \cong S$. Similarly, a *homomorphic SM* finds all subgraphs $S \subset G$ to which $P$ is homomorphic. Edge-induced SM is also called non-induced or monomorphism [16] SM. Vertex-induced SM is also called induced SM. Different from existing works [6], [11], [12] where SM refers to the edge-induced case, our SM refers to all aforementioned three cases, called *variants* and we will distinguish them when necessary. Each subgraph $S$ in the result is also called a match [20] or an embedding [11]. If a subgraph $S'$ matches a subpattern $P' \subset P$, then $S'$ is a *partial embedding* [23]. In Fig. 1, the edge-induced SM results contain both $S_1$ and $S_2$, but the vertex-induced variant contains only $S_1$. Table I shows frequently used notations.

TABLE I: Frequently used notations

| Notation | Definition |
|---|---|
| $\theta$ | a subgraph matching (SM) variant, *e.g.*, edge-induced |
| $\sigma_e$ | the label of an edge $e$ |
| $u_i \rightarrow u_j$ | also $(u_i, u_j)$, a directed edge from $u_i$ to $u_j$ |
| $u_i \mapsto v_x$ | the embedding of $u_i$ is $v_x$ |
| $G_C$ | the set of all clustered CSRs (CCSRs) of $G$ |
| $G_C^*$ | the subset of $G_C$ given $\theta$, $P$ |
| $I_R$, $I_C$ | the row index, column index arrays of a CSR |
| $f$, $f'$ | an embedding, also a mapping function $V_P \mapsto V_G$ |
| $C(u_i \vert \Phi, f)$ | data graph vertices as candidate embeddings of $u_i$ |
| $\Phi$, $\phi_k$ | a matching order, an array of pattern vertices |
| $\Phi^*$ | our final optimized matching order |

**Execution.** SM is computationally expensive. Therefore we first introduce the types of execution frameworks commonly used: backtracking [6], [16], [26] and join [11], [13]. The state-of-the-art methods of both types first find a matching order $\Phi$ of pattern vertices, *e.g.*, $u_1, \ldots, u_8$ for $P$ in Fig. 1, and then compute data graph vertices (edges) as candidate sets for pattern vertices (edges). To find results, both frameworks grow partial embeddings, *e.g.*, $(u_1, u_2) \mapsto (v_1, v_2)$, by adding $u_{x+1} \mapsto v_{x+1}$ to them following $\Phi$.

The major difference between the above two frameworks is the data they compute. Backtracking computes candidate vertices for the pattern and assembles embeddings by traversing the neighbor edges. For example, with a partial embedding $(u_1, u_2) \mapsto (v_1, v_2)$, traversing neighbor edges, *e.g.*, $(v_1, v_3)$ of $v_1$, can find candidates, *e.g.*, $u_3 \mapsto v_3$. This framework continues growing the partial embedding $(u_1, u_2, u_3) \mapsto (v_1, v_2, v_3)$ by finding candidates for $u_4$, $u_5$, etc. After finding all embeddings grown from $v_1, v_2, v_3$, the framework backtracks to $u_3$, updates to $u_3 \mapsto v_{10}$ and repeats the growing process using $v_1, v_2, v_{10}$ to find all embeddings.

In comparison, the join framework computes candidate edges, called relations, *e.g.*, denoted as $R_1(u_1, u_2)$, for all pattern edges, *e.g.*, $(u_1, u_2)$. Recent advances in worst-case optimal join (WCOJ) [27] show that we should join one vertex at a time over all edges. To find candidates of $u_1$, we join $R_1(u_1, u_2)$, $R_2(u_1, u_3)$, $R_3(u_1, u_6)$ and $R_4(u_1, u_7)$ on $u_1$. As there are multiple join operators, once we find one candidate, *e.g.*, $u_1 \mapsto v_1$, we can follow the matching order and join the next vertex $u_2$. Such a join is called a pipelined join and its advantage is shown by RapidMatch [11]. Therefore both frameworks essentially grow partial embeddings one vertex at a time by traversing the edges of candidate vertices.

TABLE II: Max pattern sizes tested in existing works.

| | |
|---|---|
| 8 or more | CFQL(32), CECI(50), Circinus(16), DAF(200), GSI(15), G-Morph(9), GuP(32), RapidMatch(32), VC(128), VEQ(200) |
| 7 or fewer | AutoMine, BENU, CliqueJoin++, cuTS, Dryadic, EdgeFrame, FlexMiner, Fractual, GF, GraphPi, GraphWCOJ, GraphZero, HUGE, LIGHT, Pangolin, Peregrine, RADS, SandSlash, STMatch, SumPA, Timely |

We survey recent works since 2019 on SM, shown in Table II. There are 21 algorithms tested for patterns with 7 or fewer vertices, but only 10 algorithms are tested for large patterns, where the numbers in parentheses show the maximal tested pattern sizes. As we focus on CPU-based single-thread in-memory SM, we do not compare with algorithms in different settings *e.g.*, GPU-based GSI [28], G-Morph [29], cuTS [30], STMatch [31], parallel or distributed HUGE [32], B-BENU [1] or requiring additional systems, *e.g.*, Spark-based EdgeFrame [33].

Before presenting our method, we introduce existing techniques for comparison. This includes existing indexing, data structures, equivalence concepts, and optimization algorithms.

**Indexing:** GADDI [34] proposes an index based on Neighboring Discriminating Substructure (NDS) distances between pairs of vertices in the data graph. It selects a set of substructures, measures their frequency within neighborhoods, and uses distance constraints to identify pattern vertex candidates. Another method, SPath [35], uses an index on the shortest paths as a neighborhood signature for each data graph vertex and matches a pattern one path at a time. However, both approaches face severe scalability issues in index construction [9], [11]. Other indexing methods, *e.g.*, gIndex [36] and C-Tree [37], are not tailored for SM [38].

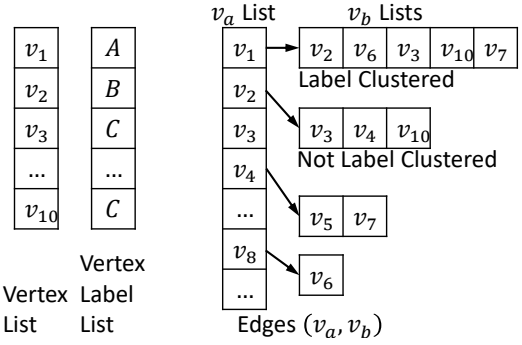


Fig. 3: Existing graph data structure of $P$ in Fig. 1.

**Data Structure.** We first introduce data structures commonly used in SM, shown in Fig. 3. RI [16], VF3 [26], Graphflow [13], RapidMatch [11] all store data graph vertices as a list, with vertex labels as another list in parallel, such that a vertex and its label share the same offset within lists.

In Fig. 3, both $v_1$ and its label $A$ are at the top position with an offset 0. Edges $(v_a, v_b)$ are stored as compressed adjacency lists, $v_a$ list and $v_b$ lists, with separate lists for the outgoing neighbors of each vertex $v_a$. The first three algorithms further store edge labels as a list in parallel with edges, the $v_b$ lists, to support edge labels. In addition, Graphflow clusters each $v_b$ list by edge labels. In Fig. 3, the outgoing neighbors of $v_1$, namely $v_3$ and $v_{10}$, are stored next to each other.

We also examine data structures used in the following works. Both GraphPi [39] and Dryadic [40] do not support either vertex labels or edge labels. VC [41], VEQ [6], DecoMine [19], Morphing [20] and GuP [14] support vertex labels but they do not present how they store the data graph. One key observation from existing works is that they support vertex labels as an extension from unlabeled graphs and therefore labels are stored in parallel with vertices.

We finally introduce data structures used in graph databases where SM is a fundamental query [42]. We should be aware that graph databases address more problems than SM, *e.g.*, graph updates, and therefore their data structures are designed to support and balance broader computation requirements. We choose one of the latest graph databases, Kùzu [17], as an example. Kùzu is read-optimized and therefore uses columnar data structures [43]. It stores vertex properties as lists [44]. Edges are stored as CSR-based adjacency list indices and edge properties are similarly stored in parallel but separate CSR-based structures. To support complex real-world data and flexible properties, graph databases often use the property graph model. In comparison, a vertex (edge) label in SM is one of the properties in the property graph model.

**Equivalence.** TurboISO [45] proposes the neighborhood equivalence class (NEC) concept to share candidate sets between pattern vertices in the same class. However, NEC only identifies a subset of equivalent pattern vertices, *e.g.*, it cannot identify all vertices of a circle pattern as one equivalence class. BoostISO [46] explores data graph vertex equivalence and proposes syntactic equivalence, *e.g.*, $v_3$, $v_{10}$ in Fig. 1, as well as query-dependent equivalence. The former is independent of the pattern and can boost all SM algorithms. However, the latter has no practical use due to its limited benefit over computation cost [47]. VEQ [6] introduces static equivalence among pattern vertices, which aligns with NEC, and dynamic equivalence among data vertices, *e.g.*, $v_3$ and $v_{10}$, to directly generate new embeddings from existing ones instead of computing them. In conclusion, existing equivalence concepts are among different vertices, enabling candidate sharing.

**Optimization.** In existing works, two types of optimization algorithms are proposed to find efficient matching orders. The first type uses heuristic rules. For example, QuickSI [48] prioritizes matching vertices associated with infrequent edges, and VF3-Light [49] selects each pattern vertex that matches the highest number of edges, has the lowest vertex label frequency, and the highest degree. RapidMatch [11] prioritizes pattern vertices connecting the highest number of matched pattern vertices. The second type is based on systematic cost estimation. For example, Graphflow [13] systematically enumerates pattern vertex permutations as candidates for matching orders. It then estimates the computation cost for each candidate, and finally selects the lowest cost one as the matching order.

## III. Motivations and Method Overview

To build an index for graph heterogeneity, we propose to cluster the data graph $G$, such that heterogeneity information, *e.g.*, labels and direction, is the same within each cluster. To preserve edges of $G$, we cluster all the edges such that edges in the same cluster are isomorphic. Consequently, we match a pattern by its edges. In Fig. 1, to match the edge $(u_1, u_3)$, we look for the cluster of edges isomorphic to $(u_1, u_3)$. Therefore we directly find candidates $(v_1, v_3)$ and $(v_1, v_{10})$ and save the computation of checking labels.

To leverage independent mappings, we propose SCE to identify them. As dependencies originate from a certain plan, we first propose a heuristic to generate an efficient plan. We then propose heuristics for fine-tuning the plan to optimize reusing mappings. In Fig. 1, instead of matching $u_3$ to a specific vertex, *e.g.*, $v_3$ in the green region $R_1$, we find all mappings, called *candidates*, for $u_3$. We also match vertices in the purple region $R_2$ and find all candidates independently of $u_3$ candidates. Consequently, we save computation by matching each region only once.

Combining the above motivations, our method, shown in Fig. 2, consists of offline (red) and online (blue, orange, and green) processing stages. As our indexing is on $G$ and achieved by clustering, we aim to reduce overhead by avoiding clustering for each SM task $\theta$. Hence, we build CCSR from $G$ and produce $G_C$ to support all SM tasks. As $G_C$ is equivalent to $G$, we do not keep $G$.

The online processing contains data reading (blue), plan optimization (orange), and execution (green). To obtain statistics for plan selection, we first extract useful $G_C^*$ from $G_C$ according to $\theta$ and $P$. As real-world graphs are sparse [16], a matching order $\Phi$ that matches as many pattern edges as possible can rapidly prune candidates and lead to efficiency. Therefore we apply GCF to produce an initial $\Phi$. To fine-tune candidate reuse, we next calculate dependency and identify SCE as $H$ from $\Phi$. Leveraging SCE, we apply LDSF to generate the dependency-aware plan $\Phi^*$.

Execution (green) is also important. As suggested by existing works [11], [13], we implement a pipelined worst-case optimal join execution framework that finds all subgraph instances of $P$ from $G_C^*$. Given the optimized plan, we follow the matching order and grow partial embeddings one vertex at a time by intersecting pattern vertex candidates.

Note that we also consider several existing techniques. At the end of optimization, to share candidates among different pattern vertices during execution, we apply neighborhood equivalence class (NEC) to identify equivalent pattern vertices. However, similar to Graphflow, GuP, RapidMatch, and VEQ, we do not apply the symmetry breaking technique of GraphPi because it does not scale for large patterns (Finding 2). In execution, we do not apply failing set pruning (FSP) because our SCE is more effective and efficient (Section V).

## IV. Clustered CSR

We propose to put isomorphic data graph edges into the same cluster. There are many ways to cluster edges, leading to different numbers of clusters. Existing works put all edges into a single cluster. As $u_2$ is a neighbor of $u_1$ and $v_1$ is a candidate of $u_1$, to find candidates of $u_2$, existing works visit all neighbors of $v_1$ and prune invalid candidates $v_3$, $v_7$, and $v_{10}$. Another choice is to put isomorphic edges, *e.g.*, $(v_1, v_2)$ and $(v_1, v_6)$, into the same cluster. This cluster is better because all neighbors of $v_1$ match $u_2$ on labels and the edge direction. A third option is to break the previous clusters into smaller ones by considering additional neighbor information. Then $(v_1, v_6)$ is excluded from the previous cluster because a candidate vertex for $u_2$ should have at least 2 outgoing neighbors but $v_6$ does not. A smaller cluster gives us more accurate candidates and efficiency, but this cluster depends on the pattern and its usefulness is limited. Therefore we use isomorphic edges to form clusters and an edge belongs to exactly one cluster.

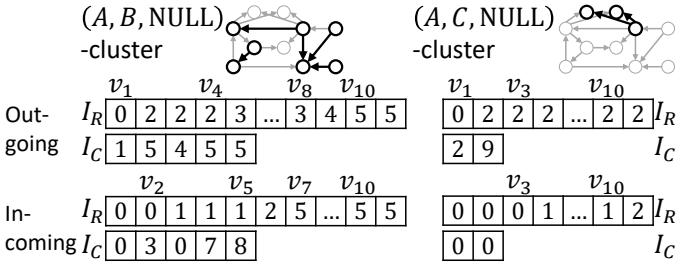


Fig. 4: Two edge isomorphism clusters in CSR for $G$ in Fig. 1.

We propose to use the compressed sparse row (CSR) to store edges in a cluster because a cluster can be viewed as a graph and CSR is an efficient data structure representing the adjacency matrix of the graph. Fig. 4 shows two clusters of $G$ in Fig. 1. A CSR stores edges, *e.g.*, $(v_i, v_j)$, in two arrays, a row index $I_R$ for vertices, and a column index $I_C$ for neighbors. To support finding both incoming and outgoing neighbors of vertices, we build two CSRs in a cluster of directed edges $v_i \to v_j$. The outgoing (incoming) CSR stores $v_i$'s outgoing ($v_j$'s incoming) neighbors in $I_C$. As we store an undirected edge $v_i - v_j$ as $(v_i, v_j)$ and $(v_j, v_i)$, we can find neighbors in either direction by building only one CSR for each undirected edge cluster. Therefore, each edge of $G$ is stored twice in exactly one cluster.

A CSR uses consecutive integers starting from 0 to represent vertices, *e.g.*, $(i-1)$ represents $v_i$. Consider the outgoing CSR in Fig. 4 left cluster, given $v_i$ and its integer representation $(i-1)$, its outgoing neighbors are $v_j$, $j - 1 = I_C[k]$, $I_R[i-1] \le k < I_R[i]$. If $i = 1$, then $0 \le k < 2$ and $v_1$ has outgoing neighbors $v_2$ and $v_6$. Note that the length $|I_C|$ is the same in outgoing and incoming CSRs. It is the number of edges, which is also the cluster size.

Compared with an adjacency list, *e.g.*, Fig. 3, or a sort trie [27], CSR arrays can search neighbors of a specific vertex in constant time, but others need a more expensive binary search. Compared with a hash trie, CSR arrays have better cache locality [27].

To retrieve the correct cluster serving as indexing, we define cluster identifiers. As edge isomorphism depends on labels of both vertices, the edge label, and the edge direction, we propose them as the cluster identifier. The identifier of a directed edge cluster arranges vertex labels in the outgoing direction. For example, $(A, B, \text{NULL})$-cluster is for the left cluster in Fig. 4, where vertices of label $A$ have outgoing edges to vertices of label $B$ and NULL means no edge labels. An undirected edge cluster uses pairs of vertex labels in both directions, sorted alphabetically, *e.g.*, $(A, B, \text{NULL})$, $(B, A, \text{NULL})$-cluster.

To show our CCSR is not expensive, we now analyze its complexity and begin with the space complexity of clusters. Given a data graph $G$ of $|V_G|$ vertices and $|E_G|$ edges, all edges, no matter whether directed or undirected, are in exactly one cluster and stored twice. Therefore the total number of edges over all clusters is $2|E_G|$. Fig. 4 shows a standard CSR and the total length of $I_R$ arrays is $2c\,(|V_G| + 1)$ where $c$ is the number of clusters, $c \le |E_G|$. As each edge corresponds to one number in $I_C$ arrays, the total length of $I_C$ is $2|E_G|$. Different from this standard CSR [44] that compresses row indices of multiple column indices, we further compress repeated numbers in $I_R$ arrays as a value and a repeat count. Therefore each edge corresponds to at most 2 integers in $I_R$ and the total length of $I_R$ has an upper bound as $4|E_G|$, which scales better than the standard one. When reading clusters, we decompress and construct standard CSRs to access neighbors. As each edge can be put into the correct cluster using its identifier in constant time, the clustering time complexity is $\mathcal{O}(|E_G|)$. Meanwhile, we sort edges in each cluster to build CSRs, and therefore the sorting complexity has an upper bound as $2|E_G| \log(2|E_G|)$.

**Algorithm 1:** ReadCSR($G_C$, $P$, $\theta$)

**Input :** All clusters $G_C$ of $G$, a pattern $P$, and the SM variant $\theta$.
**Output:** Useful CSR clusters $G_C^*$.

$G_C^* \leftarrow \emptyset$;
**foreach** $e = (u_x, u_y)$, $e \in E_P$ **do**
  $x \leftarrow L_P(u_x)$, $y \leftarrow L_P(u_y)$;
  **if** $e$ is undirected **then**
    $a \leftarrow \min(x, y)$, $b = \max(x, y)$;
    $c \leftarrow (a, b, \sigma_e)$, $(b, a, \sigma_e)$-cluster;
  **else**
    $c \leftarrow (x, y, \sigma_e)$-cluster;
  **end**
  $G_C^* \leftarrow G_C^* \cup \{c\}$ read, decompress $c$ if $c \notin G_C^*$;
**end**
**if** $\theta$ is vertex-induced **then**
  **foreach** $e = (u_x, u_y)$, $e \notin E_P$, $u_x$, $u_y \in V_P$ **do**
    **foreach** $c \in (u_x, u_y)^*$-clusters **do**
      $G_C^* \leftarrow G_C^* \cup \{c\}$ read, decompress $c$ if $c \notin G_C^*$;
    **end**
  **end**
**end**
**return** $G_C^*$;

For a given pattern and an SM variant, we only need a subset of clusters, shown in Algorithm 1, and therefore the

space complexity does not change much after reconstructing standard CSRs. For an unlabeled data graph, there are at most two clusters, one for directed edges, and the other for undirected edges. For edge-induced or homomorphic SM, we need at most $|E_P|$ clusters of edges isomorphic to pattern edges. For the vertex-induced case, we need clusters of edges between unconnected pattern vertex pairs for negation [27]. The negation removes partial embeddings where data vertices are connected but the pattern vertices are not. We use $(u_x, u_y)^*$-clusters to represent all such clusters. The total number of unconnected pattern vertex pairs is $h = \frac{1}{2}|V_P||V_P - 1| - |E_P|$, and let $\beta$ be the maximal number of negation clusters of a pair, then we need at most $(\beta h + |E_P|)$ clusters.

Note that in join approaches, *e.g.*, RapidMatch [11], for each pattern edge $e$, they create a relation that contains all data graph edges that match $e$. This is similar to our clustering idea if we view each relation as a cluster. However, they do not further use relations to identify sequential candidate equivalence and reduce repetitive matching, as discussed next.

## V. Sequential Candidate Equivalence

We propose sequential candidate equivalence (SCE) in this section. We start with candidate dependencies, and then define SCE with the help of independencies. We next propose algorithms to build the dependency and find SCE for each SM variant. We finally show SCE advantages.

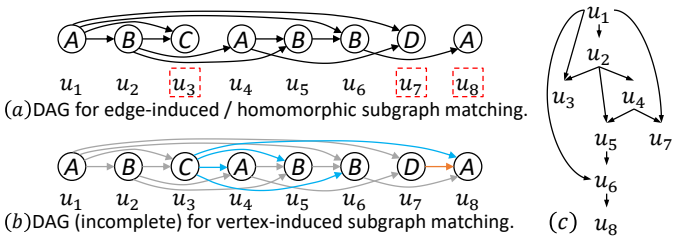


Fig. 5: Directed acyclic graphs $(a)$, $(b)$ for Fig. 1 example, and $(c)$ a different layout of $(a)$.

We use an unlabeled directed acyclic graph (DAG) to represent candidate dependencies among pattern vertices. Given a matching order, *e.g.*, $\Phi_1 = u_1, \ldots, u_8$, candidates of a pattern vertex $u_i$ can depend on candidates of another pattern vertex $u_j$ and we show this dependency as a directed edge $(u_i, u_j)$, *e.g.*, $(u_1, u_2)$. Then all dependencies form a directed graph of pattern vertices. As the matching order is a pattern vertex sequence, there is no circular dependency and the graph is acyclic. Fig. 5 shows DAGs given $\Phi_1$ of Fig. 1 $P$ for different SM variants. Both DAF [12] and VEQ [6] use DAGs, but they use DAGs to find a matching order. In contrast, we use DAGs for dependencies and find sequential candidate equivalence.

We then formally define SCE by a DAG. Let $C(u_i|\Phi, f)$ be the candidate vertex set for a pattern vertex $u_i$ given a matching order $\Phi$ and a partial embedding $f$. As an edge represents a dependency and the dependency is transitive, candidates for two pattern vertices are dependent if one, *e.g.*, $u_1$, can reach the other, *e.g.*, $u_2$, following a path, *e.g.*, an edge $(u_1, u_2)$ or edges $(u_1, u_i), (u_i, u_j), \ldots, (u_z, u_2)$. For example, $C(u_2|\Phi_1, \{u_1 \mapsto v_1\}) = \{v_2, v_6\}$ and $C(u_2|\Phi_1, \{u_1 \mapsto v_4\}) = \{v_5\}$. If there is no path in between, *e.g.*, $u_3$ and $u_4$, then two candidate sets are conditionally independent, *e.g.*, $C(u_4|\Phi_1, f_2) = C(u_4|\Phi_1, f_2 \cup \{u_3 \mapsto v_3\}) = \{v_4\}$, where $f_2 = \{u_1 \mapsto v_1, u_2 \mapsto v_2\}$. The independency further reveals the equivalence between two candidate vertex sets, and formally,

**Definition 1** (Sequential Candidate Equivalence)**.** Given the directed acyclic graph $H = (V_H, E_H)$ of candidate dependency under a matching order $\Phi$, $u_i, u_j \in V_H$, let $f$ be a partial embedding without $u_j$, if there is no path between $u_i$ and $u_j$ in $H$, then $C(u_i|\Phi, f) = C(u_i|\Phi, f \cup \{u_j \mapsto v_x\})$ in homomorphic SM, and $C(u_i|\Phi, f) \setminus \{v_x\} = C(u_i|\Phi, f \cup \{u_j \mapsto v_x\})$ in edge-induced or vertex-induced SM, where $v_x \in C(u_j|\Phi, f)$.

The edge-induced or vertex-induced SM requires $C(u_i|\Phi, f) \setminus \{v_x\}$ because $f(u_i) \neq f(u_j)$. Furthermore, if $u_i$ and $u_j$ have different labels, then $v_x \notin C(u_i|\Phi, f)$ and $C(u_i|\Phi, f) \setminus \{v_x\} = C(u_i|\Phi, f)$. We also omit $\Phi$ and $f$ when possible. Note that SCE is the candidate equivalence for the same pattern vertex over different partial embeddings, but existing candidate equivalence or vertex equivalence is between different vertices, *e.g.*, $v_3$ and $v_{10}$ [46].

**Algorithm 2:** BuildDAG($G^*_C$, $P$, $\Phi$, $\theta$)

**Input :** Clusters $G^*_C$ of a data graph, a pattern $P$, its matching order $\Phi$ and the SM variant $\theta$.
**Output:** The DAG $H$ for candidate dependency.

```
1  H = (V_H, E_H), V_H ← V_P, E_H ← ∅;
2  foreach j, 0 < j < |Φ| do
3      foreach i, 0 ≤ i < j do
4          if Neighbor(P, Φ[i], Φ[j]) then
5              E_H ← E_H ∪ {(Φ[i], Φ[j])};
6          else if θ is vertex-induced then
7              if ∃k, 0 ≤ k < i, s.t. Neighbor(P, Φ[k], Φ[j]) then
8                  if ∃α ∈ (Φ[i], Φ[j])*-clusters, s.t. |α| > 0 then
9                      E_H ← E_H ∪ {(Φ[i], Φ[j])};
10                 end
11             end
12         end
13     end
14 end
15 return H;
```

We next propose an algorithm to build the DAG $H$ for all SM variants. Algorithm 2 shows the pseudo-code. If $\Phi[i]$ and $\Phi[j]$ are neighbors in $P$, then a path is in between. Therefore, candidates $C(\Phi[j])$ depend on $C(\Phi[i])$, $i < j$, and we show this dependency by adding a directed edge $(\Phi[i], \Phi[j])$ to $H$ (Line 5). As the vertex-induced variant preserves all edges in the data graph $G$, achieving isomorphism requires removing from $C(\Phi[j])$ any neighbors of $C(\Phi[i])$ when $\Phi[i]$ and $\Phi[j]$ are not neighbors in $P$. This adds dependency between $\Phi[i]$ and $\Phi[j]$ (Line 9). Fig. 5 $(b)$ shows dependencies for unconnected pattern vertex pairs related to $u_1$ (empty), $u_3$ in blue and $u_7$ in orange, in addition to all pattern edge dependencies in $(a)$ in gray. In vertex-induced SM, each vertex $u_i$ in $H$ should have $(|V_P| - 1)$ edges because there are always direct dependencies regardless of edges between $u_i$ and other pattern vertices. However, there are two more conditions. First,

$C(\Phi[j])$ exists, computed as neighbors of $C(\Phi[k])$ (Line 7). For example, $u_1$ does not connect $u_4$, $u_5$ or $u_8$ in $H$ because their candidates do not exist. Second, there are edges in $G$ between $C(\Phi[i])$ and $C(\Phi[j])$ (Line 8). We use $(\Phi[i], \Phi[j])^*$-clusters to represent all clusters of directed and undirected edges between $\Phi[i]$ and $\Phi[j]$ regardless of edge labels. As label $D$ vertices only connect label $A$ vertices in $G$, $u_7$ only connects other label $A$ vertices in $H$.

We now analyze the space and time complexity of Algorithm 2. According to Line 6, we analyze the algorithm in two cases. The first case is edge-induced or homomorphic SM and the second case is vertex-induced SM. As the algorithm builds a DAG $H$ and we represent edges of $H$ as a hash map by mapping each vertex to its outgoing neighbor vertex set, the space complexity is the number of edges. Let $P = (V_P, E_P)$ be a pattern graph, $n = |V_P|$. In the first case, the number of edges in $H$ is the same as in $P$ because of a one-to-one mapping (Lines 4 and 5). Therefore the space complexity is $\mathcal{O}(|E_P|)$. The time complexity depends on the number of loops, which is $\mathcal{O}(n^2)$ (Lines 3 and 4). The second case can build edges in $H$ for all vertex pairs in $P$ and therefore the space complexity is $\mathcal{O}(n^2)$. Line 7 loops up to $(n-2)$ times and Line 8 can be computed in constant time because we only build non-empty clusters. Therefore the time complexity is $\mathcal{O}(n^3)$ by combining 3 nested loops of Lines 2, 3 and 7.

We finally discuss the SCE advantage using the DAG shown in Fig. 5 $(a)$. First, given a partial embedding of $u_1$ and $u_2$, if we can (or cannot) find embeddings for $u_4, \ldots, u_8$ under one mapping of $u_3$, then our findings apply to all mappings of $u_3$. Therefore, embeddings of independent vertices will either all succeed or fail in the same way, allowing for the reuse of results. Second, leveraging SCE to maximize reusing enables fine-tuning of plans.

Our SCE is more effective and efficient than the failing set pruning (FSP) technique proposed in DAF [12] and applied in RapidMatch [11] and VEQ [6]. FSP examines *failures*, partial embeddings that cannot result in an embedding, and finds pruning conditions case by case during execution. In contrast, SCE, by reusing candidates, achieves the same pruning when candidates are empty. Meanwhile, SCE is computed only once during optimization by candidate independence and is more efficient as shown in Finding 3. Notably, SCE remains effective in homomorphism, but FSP considers mapping multiple pattern vertices to the same data vertex as failures.

## VI. Optimization

There are two types of optimization algorithms to find matching orders, systematical search and heuristic rules. As the number of different matching orders increases factorially with the pattern vertex count, *e.g.*, more than $10^{18}$ matching orders for 20 vertices, systematic search becomes very expensive. In contrast, the latter only considers a small number of orders and therefore can be efficient. Although the latter does not guarantee to find the optimal matching order, heuristic rules proposed in RI [16] overall find the most efficient one, as Fig. 11 in [9] suggests. Inspired by existing works, we first introduce the idea and heuristic rules of RI [16]. Then we can propose our improvement.

As real-world graphs are sparse and most vertex pairs are not connected, matching pattern edges first can greatly prune candidates and lead to an efficient plan. Therefore RI proposes a Greatest-Constraint-First (GCF) heuristic that finds a matching order using three rules. In Fig. 1, let $P$ be the pattern graph, $\phi_k$ be the result matching order of $k$ pattern vertices, $\langle u_i, u_j \rangle \in E_P$ denote $u_i$ and $u_j$ are connected in $P$ with the edge direction ignored. First, RI chooses one of the pattern vertices, *e.g.*, $u_1$ and $u_2$, of the highest degree randomly as the first vertex in the matching order, *e.g.*, $\phi_1 = [u_1]$. Let $u_i \in \phi_k$, $u_j \notin \phi_k$, then for each pattern vertex $u_x \notin \phi_k$, RI counts vertices in the following three sets, as

$$
\begin{aligned}
T_x^1 &= \{u_i \mid \langle u_i, u_x \rangle \in E_P, 0 \le i < k\} \\
T_x^2 &= \{u_j \mid \exists u_j, \langle u_i, u_j \rangle \in E_P, \langle u_x, u_j \rangle \in E_P, 0 \le i < k\} \\
T_x^3 &= \{u_j \mid \exists u_j, \langle u_x, u_j \rangle \in E_P, \forall 0 \le i < k, \langle u_i, u_j \rangle \notin E_P\}
\end{aligned} \tag{1}
$$

and selects the vertex with the highest size. Superscripts distinguish sets and subscripts show imply the vertex to select. If there is a tie for $|T_x^1|$, then RI breaks ties by the highest $|T_x^2|$ and then $|T_x^3|$, *e.g.*, selects $u_2$ because $|T_2^3| = 2$ is the highest when $\phi_1$. RI rules can easily lead to a tie. For example, when $\phi_3 = [u_1, u_2, u_3]$, $u_4$ and $u_5$ tie for all three rules with values 1, 2 and 0. Then RI breaks the tie by selecting $u_4$.

**CCSR to break ties.** We improve RI using our CCSR. One limitation of RI is that it does not consider the data graph. As a result, given a pattern, RI generates the same matching order for different data graphs. Meanwhile, there are many ties, *e.g.*, both $u_1$ and $u_2$ have the highest degree, $u_4$ and $u_5$ tie when $\phi_3 = [u_1, u_2, u_3]$. Therefore we propose to consider the data graph to break ties using our CCSR. We choose $u_x$ such that it results in the least number of candidates. As we compute candidates by intersecting neighbors for all SM variants, we use the minimum size of corresponding clusters to estimate the number of candidates.

Concretely, if there is a tie for $\phi_1$, to minimize the number of candidates, we compute $\min(|\alpha_i|)$ for each $u_x$, where $\alpha_i$ is a cluster of edges isomorphic to $e$, $u_x \in e$, $e \in E_P$ for all $i$, and choose $u_x$ with the smallest value. If there is a tie due to $T_x^1$, let vertices $u_i$ in $\phi_{x-1}$ be neighbors of $u_x$, then for each $u_x$ in the tie, we find the smallest edge count $\omega = |I_C(u_i, u_x)|$ of all clusters isomorphic to the edge between $u_i$ and $u_x$ in $P$. We choose $u_x^*$ with the smallest $\omega$. For ties on $T_x^2$ or $T_x^3$, we similarly find $u_x^*$ as the $T_x^1$ case, but we use clusters of edges isomorphic to the edge between $u_x$ and $u_j$ in $P$, where $u_j$ satisfies $T_x^2$ or $T_x^3$. Formally we have

$$
\begin{aligned}
\omega_x^1 &= \min(\{|I_C(u_i, u_x)|\}),\ u_i \in T_x^1 \\
\omega_x^2 &= \min(\{|I_C(u_x, u_j)|\}),\ u_j \in T_x^2 \\
\omega_x^3 &= \min(\{|I_C(u_x, u_j)|\}),\ u_j \in T_x^3
\end{aligned} \tag{2}
$$

where $I_C(u_x, u_y)$ is the column index array in the cluster of edges isomorphic to that between $u_x$ and $u_y$.

**SCE to fine-tune plans.** To maximize candidate reuse, we further improve matching orders by SCE due to its advantage

shown at the end of Section V. Although a matching order defines a DAG $H$ and $H$ defines SCE, SCE can be used to improve the matching order because different matching orders can yield the same DAG, *e.g.*, in the edge-induced variant, a matching order $\Phi_2 = u_1, \ldots, u_6, u_8, u_7$ builds the same DAG as Fig. 5 $(a)$ using $\Phi_1$. Meanwhile, if we visit vertices of $H$ such that we visit all incoming neighbors of $u_x$ before visiting $u_x$, then both $\Phi_1$ and $\Phi_2$ can be visiting orders, called *topological orders* (TOs). Therefore, we want to find a matching order that is a TO of $H$ and reuses more candidates.

**Algorithm 3:** ComputeDescendant($H$)

**Input :** The DAG $H$ for candidate dependency.
**Output:** A list $A_S$, $A_S[i]$ is the descendant size of vertex $i$.

$H_C \leftarrow \text{Copy}(H)$, $A_D \leftarrow$ list of $\emptyset$, $A_P \leftarrow \emptyset$;
$A_C \leftarrow \text{VertexListWithNoChildren}(H)$;
**foreach** $j$, $j \in A_C$ **do**
  $A_D[j] \leftarrow \cup A_D[i]$, $i \in \text{OutgoingNeighbor}(H, j)$;
  **foreach** $i$, $i \in \text{IncomingNeighbor}(H_C, j)$ **do**
    remove edge $(i, j)$ from $H_C$;
    **if** $\text{OutgoingNeighbor}(H_C, i)$ is empty **then**
      Add $i$ into $A_P$;
    **end**
  **end**
  $\text{Swap}(A_C, A_P)$, $A_P \leftarrow \emptyset$;
**end**
$A_S[i] \leftarrow \text{Count}(A_D[i])$, $0 \leq i < |A_D|$;
**return** $A_S$;

We propose a Largest-Descendant-Size-First (LDSF) heuristic to select a topological order (TO) as the matching order. In $H$, each vertex has outgoing edges to other vertices, called children, that can follow immediately after in a TO. Meanwhile, a child can have multiple parents. As $H$ has no cycles, we can define the *descendant size* of a vertex $u_x$ as the number of all direct and indirect child vertices. Then this size measures the number of mappings added to $f$ that depends on $u_x$. When growing partial embeddings using $\Phi$, we want to reuse the most vertex mappings to minimize repetitive matching. In case of ties in TO, we select $u_x$ with the largest descendant size. As vertices can share descendants, we propose a dynamic programming algorithm to efficiently compute descendant sizes starting from children, shown in Algorithm 3. If there are still ties, we break them to minimize candidate counts in two steps. First, for each candidate $u_x$, we compute its minimal cluster size for all clusters of edges isomorphic to $(\Phi[i], \Phi[x]) \in P$, $i < x$. We then select $u_x$ with the smallest size. Second, among the remaining ties, we select $u_x$ with the lowest label frequency. In Fig. 5 $(c)$, $u_3$, $u_7$ and $u_8$ have the same descendant of size 1, the smallest cluster sizes as 2, 2 and 5, and the lowest label frequencies as 2, 1 and 4 respectively. Therefore $\Phi[5] = u_7$, $\Phi[6] = u_3$ and $\Phi[7] = u_8$. Algorithm 4 shows the optimization. Different from Kahn's algorithm [50] that finds a random TO, our algorithm finds a specific TO that maximizes descendant sizes and reduces candidate counts.

## VII. Experiment

**Baselines.** To compare performance, we choose state-of-the-art algorithms as baselines based on their supported SM variant and graph types. Meanwhile, we focus on all variants and different types of graphs, but not all algorithms support all cases. Therefore we identify applicable cases of baselines in Table III. Our work, CSCE, is also included in the last row. We use E (edge-induced), H (homomorphic), or V (vertex-induced) to show the variant that each algorithm supports. We use Yes (support) or No (no support) to indicate whether an algorithm supports vertex labels and edge labels. We use U (undirected) or D (directed) to show the edge direction that an algorithm supports. We use up to $x$ to show the largest pattern used in experiments presented in their papers.

**Algorithm 4:** GeneratePlan($H$, $A_S$, $G_C^*$)

**Input :** The DAG $H$, the descendant size list $A_S$, clusters $G_C^*$.
**Output:** The matching order $\Phi^*$ as a list of pattern vertices.

$Q \leftarrow \text{PriorityQueue}(A_S, G_C^*)$ that ranks vertices heuristically;
$\text{Insert}(Q, \text{AllVertexWithNoIncomingNeighbor}(H))$;
**while** $Q$ is not empty **do**
  $i \leftarrow \text{PopFirstVertex}(Q)$;
  $\text{Append}(\Phi^*, i)$;
  **foreach** $j$, $j \in \text{OutgoingNeighbor}(H, i)$ **do**
    remove edge $(i, j)$ from $H$;
    **if** $\text{IncomingNeighbor}(H, i)$ is empty **then**
      Add $i$ into $Q$;
    **end**
  **end**
**end**
**return** $\Phi^*$;

TABLE III: Algorithms to be compared.

| Algorithm | Subgraph Matching Variant | Vertex Labels | Edge Labels | Edge Direction | Pattern Size |
|---|---|---|---|---|---|
| GraphPi | E | No | No | U | Up to 7 |
| Graphflow (GF) | H | Yes | Yes | D | Up to 7 |
| GuP | E | Yes | No | U | Up to 32 |
| RapidMatch (RM) | E | Yes | No | U | Up to 32 |
| VEQ | E | Yes | No | U | Up to 200 |
| VF3 | V | Yes | Yes | U and D | Up to 2000 |
| CSCE | E, H and V | Yes | Yes | U and D | Up to 2000 |

Concretely, we choose the following algorithms as baselines. GraphPi [39] is a backtracking algorithm and it beats GraphZero [51] and Fractal [52]. Graphflow (GF) [53] is a WCOJ algorithm that beats EH [54]. GuP [14] is a backtracking algorithm that beats DAF [12], GQL-G, and GQL-R [9]. RapidMatch (RM) [11] is a pipelined WCOJ algorithm and it beats CFL-Match [55] and DAF. VEQ [6], [56] is a backtracking algorithm and it beats CFL-Match, DAF, Rifs [9], GQLfs [9] and Glasgow [57]. Finally, VF3 [26] is a backtracking algorithm and it beats L2G [58], LAD [59], RI [16] and VF2 [60].

**Metric.** We measure the total time cost in seconds (s) and the peak memory (RAM) cost in gigabytes (GB) because SM can be expensive due to a large number of embeddings. We also set a time limit of $10^4$ seconds and a memory cost limit of 192 GB, which are comparable with the chosen algorithms, *e.g.*, Graphflow uses 30 minutes and 512 GB, RapidMatch uses 5 minutes and 64 GB, and VEQ uses 10 minutes and 256 GB.

Using the given resources, SM can be unsolvable. Therefore if an algorithm fails for a pattern on a data graph, we record the time cost equal to the time limit and the observed peak memory cost following the convention in existing works [9].

TABLE IV: Dataset statistics.

| Data Graph | Edge Direction | Vertex Count | Edge Count | Label Count | Average Degree | Max In Degree | Max Out Degree |
|---|---|---|---|---|---|---|---|
| DIP | U | 4,935 | 21,975 | 0 | 8.9 | 277 | 277 |
| Yeast | U | 3,101 | 12,519 | 71 | 8.1 | 168 | 168 |
| Human | U | 4,674 | 86,282 | 44 | 36.9 | 771 | 771 |
| HPRD | U | 9,303 | 34,998 | 304 | 7.5 | 247 | 247 |
| RoadCA | U | 1,965,206 | 2,766,607 | 0 | 2.8 | 12 | 12 |
| Orkut | U | 3,072,441 | 117,185,083 | 50 | 76.3 | 33,313 | 33,313 |
| Patent | U | 3,774,768 | 33,037,894 | 20 | 8.8 | 793 | 793 |
| Subcategory | D | 2,745,763 | 13,965,410 | 36 | 10.2 | 776 | 760 |
| LiveJournal | D | 3,997,962 | 34,681,189 | 0 | 17.3 | 2,454 | 14,703 |

**Data Graphs.** We present data graphs used in experiments in Table IV. We vary both the sizes and the types of graphs to measure how algorithms scale and support different SM variants. All graphs are publicly available real-world graphs in different domains used by existing works, including protein-protein interaction networks DIP [5], HPRD and Yeast from VEQ, Human from RM, two citation graphs Patent from RM and Subcategory from GF, a road network RoadCA [61], and two online social networks LiveJournal from GF and Orkut from GraphPi. We use U or D to show if graph edges are undirected or directed. Although we model an undirected edge as a pair of directed edges, we count each undirected edge as one edge instead of two. The label count is the number of unique vertex labels, and an unlabeled graph has a label count of zero. Degrees are important statistics because they are the number of neighbor vertices that can potentially be candidates and the larger values indicate the more computationally expensive graphs. For undirected graphs, the max in degree and the max out degree are the same.

To find embeddings in a reasonable time limit, existing works choose to stop when a certain number of embeddings are found, *e.g.*, $10^5$ in [6], [11], [14]. In contrast, we aim for all SM variants and we need to verify the correct embeddings for different variants. Therefore we find all embeddings and skip extremely time-consuming experiments on large graphs with more than 10 million vertices, *e.g.*, UK2002 [51], [62] and Friendster [19], [31].

**Patterns.** To fairly compare with existing works, we use both their data graphs and patterns if available. DPCMNE [5] detects protein complexes from the DIP graph using two sets of known protein complexes, CYC and MIPS. To reduce confusion, we choose MIPS complexes as patterns that are large and appear at least once in DIP. Therefore patterns of certain sizes do not appear in our experiments. We use patterns from RM [11] for the Human and the Patent graphs. RM classifies patterns as dense (D) patterns with an average degree greater than two and otherwise sparse (S) ones. We used patterns from VEQ [6] for the HPRD and the Yeast graphs, which contain dense and sparse patterns using the same definition as RM. Therefore we name their patterns using D or S followed by pattern sizes. For all other graphs, we follow existing works, *e.g.*, RM, VEQ, and GuP, to generate patterns by sampling the corresponding data graphs. For each specific pattern configuration, *e.g.*, size and density if applicable, we measure the average of 10 different patterns. For completeness, we include small patterns in experiments, but they are out of the scope of this paper.

### A. Performance and Findings

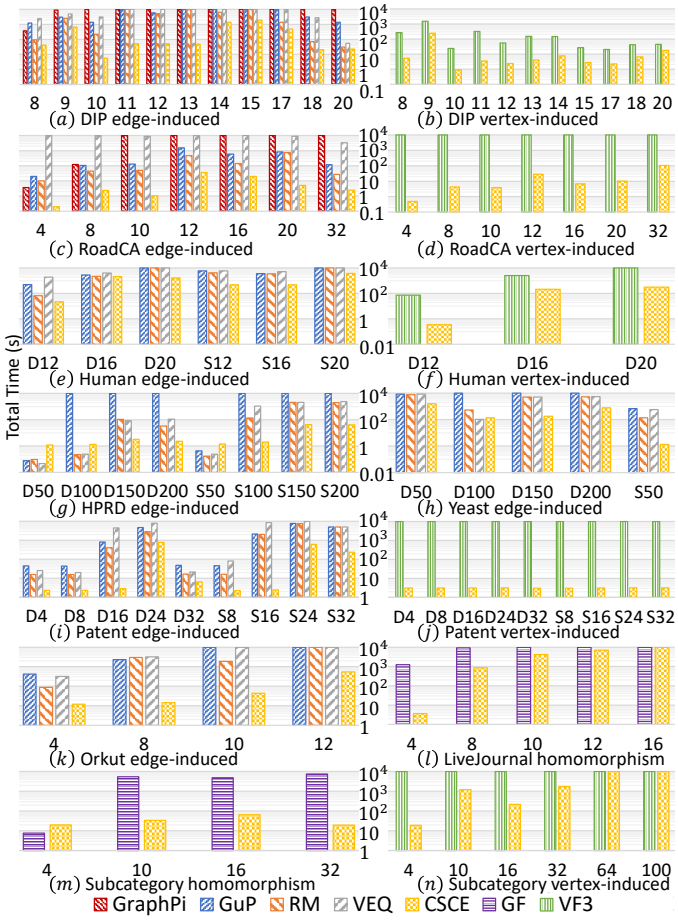


Fig. 6: Total time cost on different data and pattern graphs.

We summarize the total time to solve different SM variants in Fig. 6, including reading the data and pattern graphs, optimization, and finding all embeddings. The horizontal axis shows the pattern configuration and the vertical axis shows the total time in seconds. For undirected graphs, we do not test homomorphism because GF does not support it. For labeled graphs, we do not test GraphPi because it does not support labels. We omit the following experiments whose results are not meaningful. In the edge-induced variant, all algorithms time out on large sparse patterns in $(f)$ and $(h)$. In the vertex-induced variant, all algorithms time out on Orkut and LiveJournal, and all HPRD and Yeast patterns have 0 embeddings, which are solved in 2 seconds. **Finding 1. Except for easy cases in $(g)$ when SM takes less than 2 seconds**

**and $(m)$ when patterns are easy and small, our work is the best among all chosen algorithms up to two orders of magnitude**. As our work maintains candidate vertices for each pattern vertex and the space complexity is $\mathcal{O}(d|V_P|)$, $d$ is the average degree of $G$, our work has a low peak memory cost, less than 14 GB in all test cases.

**Undirected unlabeled graphs.** In the edge-induced variant, GraphPi explores symmetry breaking and therefore finds embeddings in a few seconds in Fig. 6 $(a)$ and $(c)$. However, optimizing symmetry breaking is expensive on unlabeled patterns with 8 or more vertices, *e.g.*, in $(a)$ on average over 450 seconds for patterns of 8 vertices and over $7,700$ seconds for 9 vertices. **Finding 2. The symmetry breaking technique does not scale on large patterns.** All of GuP, RM, and VEQ do not perform well because their techniques heavily depend on vertex labels. In contrast, our work can still benefit from SCE, though we cannot benefit from CCSR as there is only one cluster. **Finding 3. Compared with RM and VEQ, our pruning by SCE is more efficient than FSP.** In the vertex-induced variant, compared with the edge-induced variant for the same pattern, the number of result embeddings can be much smaller and the problem can be solved faster. VF3 solves all patterns efficiently because it builds indices for the data graph and looks ahead for further pattern vertices to perform pruning. Our work has a similar effect that prunes repetitive failures due to sequential candidate equivalence.

**Finding 4. Pruning by equivalent classes does not work well when graphs are sparse and unlabeled.** We also experiment with a large graph RoadCA and we show the results in Fig. 6 $(c)$. Compared with the DIP network, although the RoadCA network has more vertices, it has lower degrees and therefore finding patterns of the same size on RoadCA can be faster than DIP. This observation applies to all GraphPi, GuP, RM, and CSCE. Interestingly, VEQ often fails on small RoadCA patterns because sparse data and pattern graphs cannot put vertex into many equivalence classes and VEQ cannot benefit from the equivalences to perform pruning. Moreover, VF3 times out for graphs of millions of vertices because it is the only baseline that builds data graph indices, which is known to have scalability issues [9].

**Undirected labeled graphs.** Our work performs the best on both the Human and Yeast graphs, but it is not the best on the HPRD graph, on dense patterns of 50 and 100 vertices, and sparse patterns of 50 vertices. For patterns of all three cases, our work has an average total running time of around 1.301 seconds. Two reasons contribute to these results. First, all patterns in these three cases on average yield around 319 thousand embeddings, which are relatively infrequent compared with the average of around 3.4 billion embeddings of all other patterns. Second, we also notice that our data graph reading time, on average 0.461 seconds, is more than the total time of other baselines, *e.g.*, for dense patterns of 50 vertices, GuP, RM and VEQ have averages of 0.076, 0.096 and 0.046 seconds respectively. **Finding 5. Our CCSR has a longer reading and construction time than other data structures, but this overhead is shorter than 1 second.**

**Directed graphs.** We finally experiment with directed labeled graphs Subcategory Fig. 6 $(m)$, $(n)$ and LiveJournal $(l)$. Our work is comparable but slightly slower than GF for patterns of 4 vertices, for reasons similar to the HPRD graph. On these patterns, our work has an average data graph reading time of 16.708 seconds, which is already more than the average total time of GF of 7.670 seconds. LiveJournal has more vertices and higher degrees than Subcategory and therefore is considered more challenging. Results show that only patterns of 4, 8, 10, and 12 can be solved for the homomorphic SM.

### *B. Compare Subgraph Matching Variants*

The previous overall performances only show that our work is faster than existing works on average, but does not show how each factor affects the performance. SM performance can depend on the problem variant, the density and sizes of data and pattern graphs, and the number of embeddings in the final results. To analyze one factor and see its impact on performance, we control all other factors. As the symmetry breaking technique does not find automorphic embeddings directly, we multiply its result number of embeddings by the automorphic mappings of the pattern to reach an agreement on the number with other algorithms that do not apply symmetry breaking. To remove the impact of the number of embeddings on performances, we measure the performance by the number of embeddings per second, called *throughput* [11], and the higher the algorithm is more efficient.

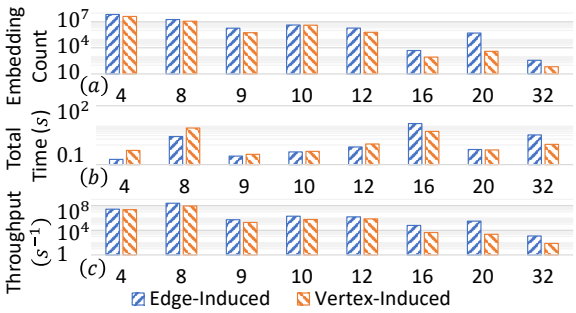


Fig. 7: Edge-induced and vertex-induced comparison.

**Finding 6. Given the same data and pattern graphs, we cannot conclude that solving the edge-induced variant takes less time than the vertex-induced variant.** As our work can solve both variants, it is interesting to learn whether one variant is easier than the other. Fig. 7 shows the numbers of embeddings $(a)$, the total time $(b)$, and the throughput $(c)$ for each variant for patterns of different sizes on the RoadCA graph. One might think the former is easier because it does not need to filter embeddings that contain more edges than the given pattern. This is not true for the total time if there are many more edge-induced embeddings than vertex-induced ones, *e.g.*, patterns of sizes 16 and 32. The latter is easier in these patterns because CSCE filters partial embeddings whenever possible but not after finding all edge-induced embeddings. However, we observe the former

has higher throughput due to skipping filtering. Therefore to efficiently solve SM, it requires significant modification to adapt an edge-induced algorithm to a vertex-induced one or two separate algorithms. Directly filtering the final edge-induced embeddings is low efficient.

**Finding 7. Given the same data and pattern graphs, we observe that the homomorphic variant can be solved faster than the vertex-induced variant.** This can be observed from experiments on LiveJournal and Subcategory graphs. We do not perform experiments on homomorphism and edge-induced variants because none of the baselines support both variants.

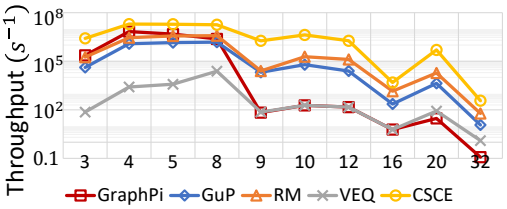


Fig. 8: Edge-induced throughput on the RoadCA graph.

**Finding 8. Matching large patterns is more difficult than smaller ones, but this trend is not strict because it also depends on the data and pattern graph topologies.** We use the RoadCA graph to measure the edge-induced variant performance as most chosen algorithms support it. Our work already shows the trend that the throughput decreases as the pattern sizes increase in Fig. 7 $(c)$, and therefore we further include existing works in the comparison. As shown in Fig. 8, it is clear that matching large patterns is more challenging, and existing works have lower throughput than our work.

### C. Scalability

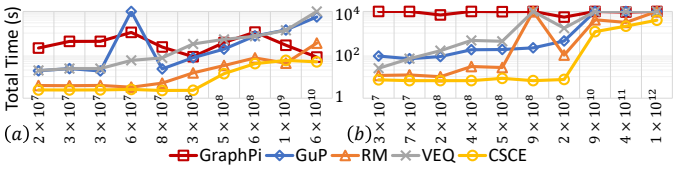


Fig. 9: Scalability by varying the number of embeddings.

As SM can yield a large number of embeddings and more embeddings can lead to longer execution time due to larger search space, we evaluate the scalability of our work by varying the number of embeddings. As shown in two subfigures in Fig. 9, we choose edge-induced SM on the DIP graph using patterns of $(a)$ 8 and $(b)$ 9 vertices as most baselines support them. We use 10 patterns for each size and arrange performances in the ascending order of the number of embeddings, shown as the horizontal axes. **Finding 9. Our work has a similar scalability as existing works as our total time increases with the number of embeddings.** However, GraphPi is an exception as its total time is between $10^2$ and $10^3$ seconds in $(a)$ and is close to $10^4$ seconds in $(b)$. This is because its optimization does not scale with the pattern size, resulting in a longer optimization time than execution. As its optimization time does not depend on the number of embeddings, the total time of GraphPi does not go up, but it is inefficient compared with other algorithms.

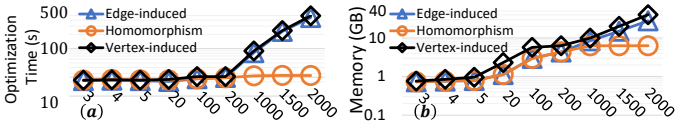


Fig. 10: Scalability by varying pattern sizes.

Although finding all embeddings can be extremely time-consuming when $G$ contains more labels and $P$ becomes larger, we can analyze how our final plan generation time scales for all SM variants. Fig. 10 shows $(a)$ the time and $(b)$ the peak memory cost on average for different pattern sizes on the Patent graph with 2000 randomly assigned vertex labels. Therefore we have **Finding 10. Our work can support patterns up to 2000 vertices within 500 seconds using less than 40 GB of memory. Meanwhile, homomorphism does not require injectivity and therefore can be optimized in 30 seconds.**

### D. CCSR Overhead

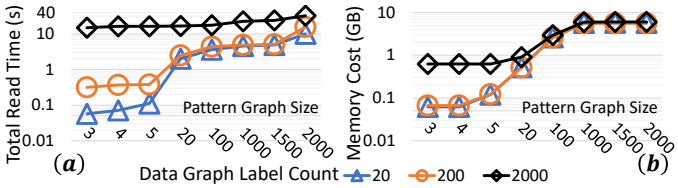


Fig. 11: CCSR overhead by varying labels and pattern sizes.

Our CCSR has a longer reading and decompression time as shown in **Finding 5**. To further measure its overhead, we show in Fig. 11 $(a)$ the reading time and $(b)$ the peak memory cost when reading different data and pattern graphs. As our CCSR overhead increases when the number of data graph labels increases, we vary the number of labels of the Patent graph to 20, 200, and 2000 labels, each represented as a trajectory. Meanwhile, we only read clusters that are used by patterns, and therefore we also vary pattern graph sizes to 3, 4 till 2000. **Finding 11. Considering the given resources, the time and space overhead of our CCSR is acceptable.**

### E. SCE Occurrence

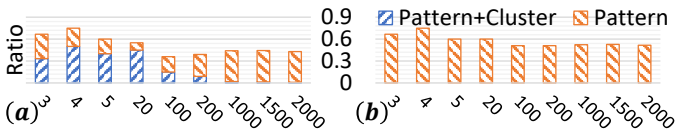


Fig. 12: SCE occurrence in different Patent pattern sizes.

Our algorithm is built on SCE and therefore we measure how often candidates of a pattern vertex are independent of other candidates and how often clustering contributes to SCE. Fig. 12 shows the average SCE occurrences as the

bar height, measured as the percent of pattern vertices that satisfy SCE in both $(a)$ edge-induced and $(b)$ homomorphic variants. Cluster sub-bars show the ratio of SCE that satisfies injectivity $C(u_i|\Phi, f) \setminus \{v_x\} = C(u_i|\Phi, f)$ due to empty clusters. Homomorphism does not require injectivity and thus has no cluster sub-bars. **Finding 12. On average 51% pattern vertices show SCE in the edge-induced variant and 58% show SCE in homomorphism. Meanwhile, the ratio of cluster contribution reduces when the pattern grows larger.** Furthermore, the vertex-induced variant preserves the absence of pattern edges in subgraph instances, and therefore all SCE found is due to clusters (Algorithm 2 Lines 7 and 8). Meanwhile, the average SCE for sparse patterns is 3.3%, but for dense patterns, the average increases as the pattern sizes increase, *e.g.*, 2.7% for 50 vertices, 4.4% for 100 vertices, 4.1% for 150 vertices and 5.0% for 200 vertices.

### F. Query Plan Quality

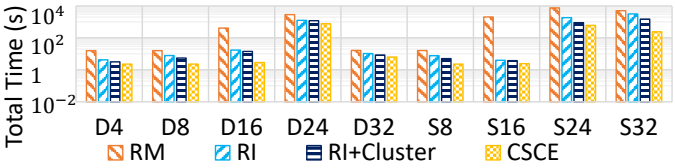


Fig. 13: Performance comparison among different query plans.

As we have shown SCE appears frequently, we want to measure how each of our optimization techniques contributes to the plan quality. According to Fig. 6, the plan of RM has the overall best performance, therefore we use it as a baseline and also use its Patent graph in comparison. In addition, we include RI, as we claim its heuristic is best, RI+Cluster as we claim using the data graph can improve the plan quality, and CSCE that further considers SCE. **Finding 13. Fig. 13 shows our plan is the best by considering clusters and SCE.**

### G. Case Study

The advantages of our approach are demonstrated through a real-world graph clustering scenario using the EMAIL-EU dataset, which captures the email communication between members in a large European research institution [61]. Our goal is to determine whether two members belong to the same department based on their communication patterns, thereby clustering members with the same department indicated by the same cluster. An edge-based clustering approach achieves an F1 score of 0.398 (Table 2 [63]). In contrast, leveraging higher-order clustering using 8-cliques, we improve the score to 0.515. Meanwhile, our method reduces the execution time from 11.57 to 0.39 seconds on finding 8-clique instances.

### H. Less Effective Scenarios

We analyze the impact of symmetry breaking. On the one hand, due to scalability concerns (Finding 2), our method, optimized for large heterogeneous patterns, does not apply symmetry breaking (Section III). On the other hand, as shown in Fig. 14 $(a)$, the improvement of symmetry breaking is

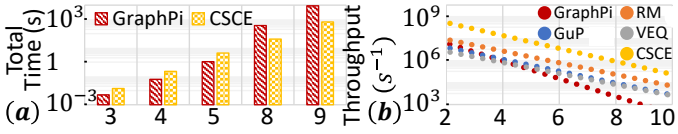


Fig. 14: DIP Performance on $(a)$ size 3, 4, 5, 8, and 9 patterns $(b)$ density of pattern sizes 8 to 20 (exponential fitting).

marginal, *e.g.*, less than 1 second on small (5 vertices or fewer) homogeneous symmetric patterns, and diminishes when patterns grow larger, *e.g.*, 8 or more vertices.

We acknowledge two less effective scenarios in our work. First, although our throughput drops on denser patterns, our work still outperforms existing approaches by throughput, as shown in Fig. 14 $(b)$. Graph density poses a common challenge in existing methods. For example, all of GuP [14], RM [11] and VEQ [6] rely on independence between unconnected pattern vertex pairs, but such pairs are less frequent as patterns become denser. Our CCSR compression is less effective on denser data graphs and SCE drops on denser patterns. Second, in Fig. 6 $(g)$ and $(m)$, CSCE falls behind on short-running patterns due to CCSR decompression overhead (Section IV). However, our overhead is short and acceptable (Finding 5).

## VIII. Conclusion

This work focuses on generating efficient plans to match large patterns. We propose CCSR to support heterogeneous graphs and SCE for dependency-aware plan optimization. Furthermore, our work, CSCE, efficiently solves three SM variants on heterogeneous graphs. Experiments show that CSCE overall outperforms all existing works on large patterns up to two orders of magnitude. Meanwhile, our optimization can scale up to patterns of 2000 vertices on graphs with millions of vertices. Future work can focus on reducing the overhead of CCSR and applying CSCE to improve matching orders using different heuristics.


## Acknowledgment

This material is based upon work supported by the National Science Foundation IIS 16-19302 and IIS 16-33755, Zhejiang University ZJU Research 083650, Futurewei Technologies HF2017060011 and 094013, IBM-Illinois Center for Cognitive Computing Systems Research (C3SR) and IBM-Illinois Discovery Accelerator Institute (IIDAI), grants from eBay and Microsoft Azure, UIUC OVCR CCIL Planning Grant 434S34, UIUC CSBS Small Grant 434C8U, and UIUC New Frontiers Initiative. Reynold Cheng, Xiaodong Li, and Matin Najafi were supported by the Hong Kong Jockey Club Charities Trust (Project 260920140), the University of Hong Kong (Project 109000579), and the HKU Outstanding Research Student Supervisor Award 2022-23. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

## REFERENCES


[1] Z. Wang, W. Hu, G. Chen, C. Yuan, R. Gu, and Y. Huang, "Towards efficient distributed subgraph enumeration via backtracking-based framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2953–2969, 2021.

[2] A. R. Benson, D. F. Gleich, and J. Leskovec, "Higher-order organization of complex networks," *Science*, vol. 353, no. 6295, pp. 163–166, 2016.

[3] V. Spirin and L. A. Mirny, "Protein complexes and functional modules in molecular networks," *Proceedings of the National Academy of Sciences*, vol. 100, no. 21, pp. 12 123–12 128, 2003.

[4] J. Chen and B. Yuan, "Detecting functional modules in the yeast protein–protein interaction network," *Bioinformatics*, vol. 22, no. 18, pp. 2283–2290, 2006.

[5] X. Meng, J. Xiang, R. Zheng, F.-X. Wu, and M. Li, "Dpcmne: detecting protein complexes from protein-protein interaction networks via multi-level network embedding," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 19, no. 3, pp. 1592–1602, 2021.

[6] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, "Versatile equivalences: Speeding up subgraph query processing and subgraph matching," in *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021, pp. 925–937.

[7] H. Ji, X. Wang, C. Shi, B. Wang, and S. Y. Philip, "Heterogeneous graph propagation network," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 521–532, 2021.

[8] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao, "Answering natural language questions by subgraph matching over knowledge graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 5, pp. 824–837, 2017.

[9] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1083–1098.

[10] S. Sun, Y. Che, L. Wang, and Q. Luo, "Efficient parallel subgraph enumeration on a single machine," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 232–243.

[11] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "Rapidmatch: a holistic approach to subgraph query processing," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 2, pp. 176–188, 2020.

[12] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, 2019, pp. 1429–1446.

[13] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 12, no. 11, pp. 1692–1704, 2019.

[14] J. Arai, Y. Fujiwara, and M. Onizuka, "Gup: Fast subgraph matching by guard-based pruning," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–26, 2023.

[15] T. Jin, B. Li, Y. Li, Q. Zhou, Q. Ma, Y. Zhao, H. Chen, and J. Cheng, "Circinus: Fast redundancy-reduced subgraph matching," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[16] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC bioinformatics*, vol. 14, no. S7, p. S13, 2013.

[17] X. Feng, G. Jin, Z. Chen, C. Liu, and S. Salihoğlu, "Kùzu graph database management system," in *The Conference on Innovative Data Systems Research*. CIDR, 2023.

[18] X. Li, R. Cheng, M. Najafi, K. Chang, X. Han, and H. Cao, "M-cypher: A gql framework supporting motifs," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management (CIKM)*, 2020, pp. 3433–3436.

[19] J. Chen and X. Qian, "Decomine: A compilation-based graph pattern mining system with pattern decomposition," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2022, pp. 47–61.

[20] K. Jamshidi, H. Xu, and K. Vora, "Accelerating graph mining systems with subgraph morphing," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 162–181.

[21] X. Li, R. Cheng, K. Chang, C. Shan, C. Ma, and H. Cao, "On analyzing graphs with motif-paths," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 6, pp. 1111–1123, 2021.

[22] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: a two-level framework for efficient graph pattern mining," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 378–391.

[23] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 581–594.

[24] V. Lipets, N. Vanetik, and E. Gudes, "Subsea: an efficient heuristic algorithm for subgraph isomorphism," *Data Mining and Knowledge Discovery*, vol. 19, no. 3, pp. 320–350, 2009.

[25] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 10, no. 3, pp. 217–228, 2016.

[26] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 804–818, 2017.

[27] T. L. Veldhuizen, "Triejoin: A simple, worst-case optimal join algorithm," in *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, N. Schweikardt, V. Christophides, and V. Leroy, Eds., 2014, pp. 96–106.

[28] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "Gsi: Gpu-friendly subgraph isomorphism," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1249–1260.

[29] B. Rowe and R. Gupta, "G-morph: Induced subgraph isomorphism search of labeled graphs on a gpu," in *European Conference on Parallel Processing*. Springer, 2021, pp. 402–417.

[30] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam, "cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[31] Y. Wei and P. Jiang, "Stmatch: accelerating graph pattern matching on gpu with stack-based loop optimizations," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–13.

[32] Z. Yang, L. Lai, X. Lin, K. Hao, and W. Zhang, "Huge: An efficient and scalable subgraph enumeration system," in *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021, pp. 2049–2062.

[33] P. Fuchs, P. Boncz, and B. Ghit, "Edgeframe: Worst-case optimal joins for graph-pattern matching in spark," in *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2020, pp. 1–11.

[34] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009, pp. 192–203.

[35] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 340–351, 2010.

[36] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 335–346.

[37] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *2006 IEEE 22nd International Conference on Data Engineering (ICDE)*. IEEE, 2006, pp. 38–38.

[38] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 2, pp. 133–144, 2012.

[39] T. Shi, M. Zhai, Y. Xu, and J. Zhai, "Graphpi: High performance graph pattern matching through effective redundancy elimination," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[40] D. Mawhirter, S. Reinehr, W. Han, N. Fields, M. Claver, C. Holmes, J. McClurg, T. Liu, and B. Wu, "Dryadic: Flexible and fast graph pattern matching at scale," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 289–303.

[41] S. Sun and Q. Luo, "Subgraph matching with effective matching order and indexing," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[42] Z. Lai, Z. Yang, and L. Lai, "Improving distribued subgraph matching algorithm on timely dataflow," in *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2019, pp. 266–273.

[43] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented dbms," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 2018, pp. 491–518.

[44] P. Gupta, A. Mhedhbi, and S. Salihoglu, "Columnar storage and list-based processing for graph database management systems," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 11, pp. 2491–2504, 2021.

[45] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 337–348.

[46] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 5, pp. 617–628, 2015.

[47] L. Zeng, Y. Jiang, W. Lu, and L. Zou, "Deep analysis on subgraph isomorphism," *arXiv preprint arXiv:2012.06802*, 2020.

[48] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 364–375, 2008.

[49] V. Carletti, P. Foggia, A. Greco, M. Vento, and V. Vigilante, "Vf3-light: A lightweight subgraph isomorphism algorithm and its experimental evaluation," *Pattern Recognition Letters*, vol. 125, pp. 591–596, 2019.

[50] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[51] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: A high-performance subgraph matching system," *ACM SIGOPS Operating Systems Review*, vol. 55, no. 1, pp. 21–37, 2021.

[52] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, 2019, pp. 1357–1374.

[53] A. Mhedhbi, C. Kankanamge, and S. Salihoglu, "Optimizing one-time and continuous subgraph queries using worst-case optimal joins," *ACM Transactions on Database Systems (TODS)*, vol. 46, no. 2, pp. 1–45, 2021.

[54] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 4, pp. 1–44, 2017.

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

[56] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, "Fast subgraph query processing and subgraph matching via static and dynamic equivalences," *The VLDB Journal*, vol. 32, no. 2, pp. 343–368, 2023.

[57] C. McCreesh, P. Prosser, and J. Trimble, "The glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants," in *Graph Transformation: 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25–26, 2020, Proceedings 13*. Springer, 2020, pp. 316–324.

[58] I. Almasri, X. Gao, and N. Fedoroff, "Quick mining of isomorphic exact large patterns from large graphs," in *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 2014, pp. 517–524.

[59] C. Solnon, "Alldifferent-based filtering for subgraph isomorphism," *Artificial Intelligence*, vol. 174, no. 12-13, pp. 850–864, 2010.

[60] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[61] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[62] C. Gui, X. Liao, L. Zheng, P. Yao, Q. Wang, and H. Jin, "Sumpa: Efficient pattern-centric graph mining with pattern abstraction," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 318–330.

[63] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2017, pp. 555–564.