Circinus: Fast Redundancy-Reduced Subgraph Matching

TATIANA JIN*, The Chinese University of Hong Kong, Hong Kong SAR and Kasma Pte Ltd., Singapore
BOYANG LI*, The Chinese University of Hong Kong, Hong Kong SAR
YICHAO LI, The Chinese University of Hong Kong, Hong Kong SAR and Kasma Pte Ltd., Singapore
QIHUI ZHOU, The Chinese University of Hong Kong, Hong Kong SAR
QIANLI MA and YUNJIAN ZHAO, The Chinese University of Hong Kong, Hong Kong SAR
HONGZHI CHEN, Kasma Pte Ltd., Singapore
JAMES CHENG, The Chinese University of Hong Kong, Hong Kong SAR

Subgraph matching is one of the most important problems in graph analytics. Many algorithms and systems have been proposed for subgraph matching. Most of these works follow Ullmann's backtracking approach as it is memory-efficient in handling an explosive number of intermediate matching results. However, they have largely overlooked an intrinsic problem of backtracking, namely repeated computation, which contributes to a large portion of the heavy computation in subgraph matching. This paper proposes a subgraph matching system, Circinus, which enables effective computation sharing by a new compression-based backtracking method. Our extensive experiments show that Circinus significantly reduces repeated computation, which transfers to up to several orders of magnitude performance improvement.

CCS Concepts: • **Information systems → Information retrieval query processing**; **Graph-based database models**.

Additional Key Words and Phrases: subgraph matching, graph database, graph mining

ACM Reference Format:
Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1, Article 12 (May 2023), 26 pages. https://doi.org/10.1145/3588692

12

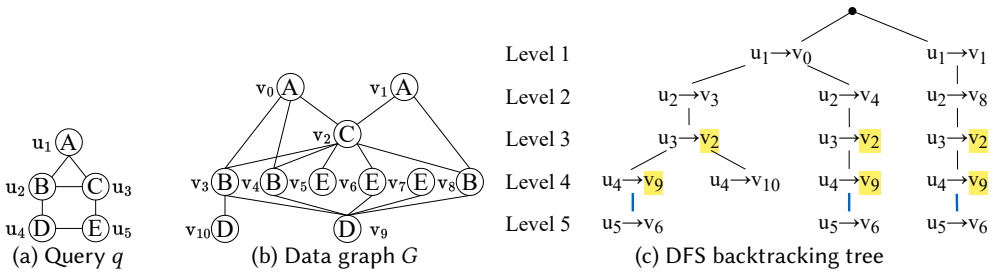
1 INTRODUCTION

Subgraph matching is a fundamental problem in graph analytics and has a wide range of applications [32] such as bio-informatics [27], chemistry [45], finance [5], and social network analysis [35]. Given a *data graph* G and a *query graph* q, where vertices in G and q may be labeled or unlabeled, the problem of subgraph matching is to find all subgraphs of G that are isomorphic to q.

*Both authors contributed equally to this research.

Authors' addresses: Tatiana Jin, tjin@cse.cuhk.edu.hk, jin@kasma.ai, The Chinese University of Hong Kong, Hong Kong, Hong Kong SAR and Kasma Pte Ltd., Singapore; Boyang Li, byli@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, Hong Kong SAR; Yichao Li, yichaoli21@cse.cuhk.edu.hk, liyichao@kasma.ai, The Chinese University of Hong Kong, Hong Kong, Hong Kong SAR and Kasma Pte Ltd., Singapore; Qihui Zhou, qhzhou@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, Hong Kong SAR; Qianli Ma, qlma@cse.cuhk.edu.hk; Yunjian Zhao, yjzhao@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, Hong Kong SAR; Hongzhi Chen, chenhongzhi@kasma.ai, Kasma Pte Ltd., Singapore; James Cheng, jcheng@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, Hong Kong SAR.

2836-6573/2023/5-ART12 $15.00
https://doi.org/10.1145/3588692



(a) Query q (b) Data graph G (c) DFS backtracking tree

Fig. 1. A backtracking example

Most existing graph querying systems [1, 6, 8, 11, 12, 17, 19, 29, 34, 37–39, 46] and algorithms [2–4, 14, 16, 18, 31, 33, 47] process a subgraph matching query by following the classic *backtracking* approach by Ullmann [40]. We focus on backtracking-based solutions in this paper.

In backtracking, *query vertices* (i.e., vertices in a query) are mapped to *data vertices* (i.e., vertices in a data graph) by following a *matching order*. The backtracking process can be regarded as a depth-first search (DFS) tree, in which any simple path from the root corresponds to a *partial/full match* of the query. For example, Figure 1 shows the DFS tree for matching a query q in a data graph G, where the matching order is $u_1 \prec u_2 \prec u_3 \prec u_4 \prec u_5$. The leftmost path in the DFS tree maps query vertices $(u_1, u_2, u_3, u_4, u_5)$ to data vertices $(v_0, v_3, v_2, v_9, v_6)$, which corresponds to a *full match* of q in G, i.e., the subgraph of G induced by $\{v_0, v_3, v_2, v_9, v_6\}$ is a match of q. The second path maps (u_1, u_2, u_3, u_4) to (v_0, v_3, v_2, v_{10}), which corresponds to a *partial match* of q, but it cannot be further extended to match u_5 (since v_{10} maps to u_4 and u_5 is a neighbor of u_4, v_{10} must have a neighbor labeled 'E' in G in order to match u_5).

Existing solutions and limitations. While graph mining systems [7, 11, 17, 24, 38, 42, 44] and graph databases [6, 10, 19, 29, 37] could be used to process a subgraph matching query, specialized subgraph matching systems [1, 8, 12, 23, 34, 46] and algorithms [2–4, 14–16, 18, 20, 30, 47] have been proposed to provide much better performance (Section 6). These specialized solutions have used various techniques to reduce redundant computation, including elimination of duplicate matches caused by graph automorphism [1, 8, 11, 12, 17, 34, 38, 39, 46], computation sharing among query vertices with equivalence relationship [15, 20, 30, 34], candidate pruning [2, 3, 14, 16, 18, 20, 47], and matching order selection to reduce backtracking search space [3, 14, 18, 33, 34].

Although the existing solutions have effectively reduced much redundant computation, there is one critical type of repeated computation in subgraph matching that has been largely overlooked.[1] *This repeated computation occurs when we are extending partial matches at the same level of the backtracking tree.* For example, in Figure 1c, there are three paths that map u_3 to v_2 and u_4 to v_9. To extend these three paths to match u_5, which is a common neighbor of u_3 and u_4, existing solutions intersect the neighbor sets of v_2 and v_9 for each of the three paths. Thus, the same neighbor set intersection operation is repeated three times. Such redundant computation is very frequent in dense clusters of a graph and in power-law graphs.

We can use massive parallelization to process the abundant set intersections to reduce query latency, e.g., using GPUs [8, 12, 13]. However, a less brute-force approach is by effectively sharing the repeated set intersections. Unfortunately, it is not easy to enable such computation sharing among different paths in a backtracking tree because the repeated intersections may not belong to paths that share the same prefix (as shown in Figure 1c), and the paths may not be executed

[1]To the best of our knowledge, only [25] reuses computation among partial matches at the same level, and only when consecutive partial matches share the same prefixes.

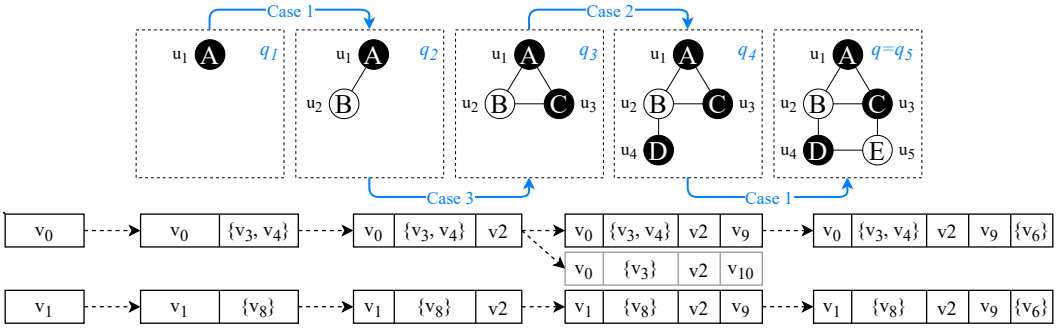


Fig. 2. An example sequence of vertex covers and compressed groups (black vertices in q_i are the VC of q_i)

consecutively. Computation sharing becomes even harder when parallelization takes place, because different paths may share the same set intersection and these paths may be processed by different threads at different time. The problems of non-consecutive occurrences and parallelization also make caching of intersection results ineffective. We may also eliminate such redundancy using a BFS approach, by grouping partial matches that share the same intersection and are at the same level of the search tree, but the memory overhead of BFS is usually prohibitive [7, 8, 38, 42]. Thus, *a new mechanism is needed to enable computation sharing among partial matches while preserving DFS-based memory efficiency.*

In this paper, we propose a new compression-based backtracking method that generates compressed groups of partial matches, so that computation can be shared within each group. We also devise an optimization strategy to select the compression plan that gives the lowest estimated computation redundancy. To apply our compression-based backtracking method to existing solutions (e.g., CFL [3] and GQL [16]) to improve their performance, we propose a general subgraph matching framework, called *Circinus*, which includes a modular optimization workflow and an operator-based execution pipeline. We implement our compression-based backtracking method as a module in Circinus and also propose compression-specific optimizations to speed up query execution. Extensive experimental results verify that Circinus further reduces repeated computation significantly on top of existing optimization techniques. The reduced computation effectively transfers to up to several orders of magnitude performance improvement over existing methods [3, 13, 16, 17, 28, 34, 36, 41].

2 MOTIVATION

In spite of previous efforts in reducing redundant computation, an intrinsic source of redundant computation exists in the backtracking process of subgraph matching.

Observation. Consider the i-th level in a backtracking tree. For each node w at the i-th level, the root-to-w path corresponds to a partial match that has matched the first i query vertices according to a given matching order. Let u_{i+1} be the $(i+1)$-th query vertex, and u_j be a *parent query vertex* of u_{i+1} if u_j is a neighbor of u_{i+1} and ordered before u_{i+1}. In a partial match, the data vertex v_j that maps to u_j is called a *parent data vertex* of u_{i+1}. Let the set of parent data vertices of u_{i+1} be $\mathcal{P}_{i+1}$. To extend each partial match at the i-th level to match u_{i+1}, we perform an `Extend` operation that computes $\bigcap_{p \in \mathcal{P}_{i+1}} neighbors(p)$, i.e., the intersection of the sets of neighbors of all the parent data vertices of u_{i+1}. For example, in Figure 1, if $u_{i+1} = u_5$, its parent query vertices u_j are u_3 and u_4. In Figure 1c, (v_0, v_3, v_2, v_9) corresponds to a partial match of (u_1, u_2, u_3, u_4), and thus v_2 and v_9 are the parent data vertices of u_5. To extend (v_0, v_3, v_2, v_9) to match u_5, we intersect the neighbor sets of v_2

and v_9 to find their common neighbor that has the same label as u_5, which gives v_6 and hence the extended match $(v_0, v_3, v_2, v_9, v_6)$.

We can observe that the intersection only involves the neighbor sets of the parent data vertices, while the data vertices that map to other query vertices are irrelevant. This presents a computation sharing opportunity: if the partial matches at each level are grouped by the parent data vertices, they can share the intersection results when performing the `Extend` operation. Refer to Figure 1c as an example, to extend (v_0, v_3, v_2, v_9), (v_0, v_4, v_2, v_9) and (v_1, v_8, v_2, v_9) to match u_5, we can group them by (v_2, v_9) and intersect the neighbor sets of v_2 and v_9 only once to obtain v_6 (which matches u_5).

Verification. We report the computation redundancy in two seminal subgraph matching methods, CFL [3] and GQL [16], and a recent system Peregrine [17]. We measured the number of set intersection computations in each method. As a reference, we also computed the optimal number of set intersection calls for each method that is achievable by sharing computation among all partial matches at the same level of the backtracking tree. We defer the full description of the experimental setting until Section 5.3. We find that for all three solutions, the difference between the actual intersection counts and the corresponding optimal numbers can be one to two orders of magnitude. Namely, the repeated computation accounts for 90%-99% of the total set intersection computation, which renders a large space for computation sharing.

3 COMPRESSION-BASED BACKTRACKING

We propose a new compression-based backtracking method, which enables computation sharing by grouping partial matches at each level of a backtracking search tree, while still following the DFS approach for efficient memory usage.

3.1 Subgraph Compression

We first illustrate the main idea using an example. Consider the query q shown in Figure 1 and the matching order $u_1 \prec u_2 \prec u_3 \prec u_4 \prec u_5$. For $1 \leq i \leq 5$, let q_i be the subgraph of q induced by $\{u_1, u_2, \ldots, u_i\}$ as shown in Figure 2, and let M_i be the set of partial matches at Level i of the DFS tree in Figure 1, i.e., the matches of q_i. For example, M_4 consists of the four subgraphs of G induced by $\{v_0, v_3, v_2, v_9\}$, $\{v_0, v_3, v_2, v_{10}\}$, $\{v_0, v_4, v_2, v_9\}$, and $\{v_1, v_8, v_2, v_9\}$, which all match q_4 (shown in Figure 2).

Our goal is to group (*by key*) the subgraphs in each M_i to maximize computation sharing when extending M_i to M_{i+1} (i.e., to match q_{i+1} in G). Consider extending M_4 to M_5, the optimal grouping would be using the mappings of $\{u_3, u_4\}$ (i.e., $\{v_2, v_9\}$, $\{v_2, v_{10}\}$) as the key, which allows the intersection of the neighbor sets of v_2 and v_9 to be shared among three partial matches in M_4. However, this optimal grouping requires M_i to be wholly computed, which is essentially a BFS approach and is memory prohibitive.

Can we compute one group at a time for each level instead of computing all groups as in a BFS approach while still effectively sharing computation? Inspired by the compression technique in CBF[2] [28], we propose to use a vertex cover (VC) of q_i to set the grouping keys. For each M_i, the data vertices mapping to the query vertices in the vertex cover are set as the grouping key.

Consider q_4 and use $\{u_1, u_3, u_4\}$ as its VC. The data vertices mapping to (u_1, u_3, u_4) in the partial matches in M_4 are (v_0, v_2, v_9), (v_0, v_2, v_{10}), (v_0, v_2, v_9), (v_1, v_2, v_9). Thus, if we use the mapping data vertices as the key, M_4 is divided into three groups: $(v_0, \{v_3, v_4\}, v_2, v_9)$, $(v_0, \{v_3\}, v_2, v_{10})$, $(v_1, \{v_8\}, v_2, v_9)$, where (v_0, v_3, v_2, v_9) and (v_0, v_4, v_2, v_9) are compressed into a single group $(v_0, \{v_3, v_4\}, v_2, v_9)$ and can

[2]CBF uses vertex cover to compress the outputs, while we compress partial matches for computation sharing. See more details in Section 6.

be extended by sharing the intersection of the neighbors of v_2 and v_9. Note that the compression (and hence the computation sharing) does not look impressive in this toy example, but is very significant in real data graphs as shown in Figure 11 and we will show more in Section 5.3.

3.2 Compression-based Extend Operator

With VC-based grouping, we design a new `Extend` operator, which computes extensions on compressed groups and transforms the standard backtracking process into *compression-based backtracking*.

Design choices. In our design of of the `Extend` operator, we make the following three design choices to bound the memory usage and to ensure a low grouping overhead.

First, we enforce that each grouping key must be a VC of the subgraphs in the group, so that `Extend` can generate one group of partial matches at each level at a time, without storing an entire level of partial matches as in BFS-based approaches. For each group of partial matches, the VC-based compression [28] avoids enumerating the cross-products for the sets in the group, i.e., for a group $(v_1, \{x_1, \ldots, x_m\}, v_2, \{y_1, \ldots, y_n\})$, without compression we need to enumerate $O(mn)$ partial matches in the form of (v_1, x_i, v_2, y_j). Thus, the VC-based compression reduces the memory overhead. In particular, the memory required for a compressed group at each step of backtracking is bounded by the total number of mapping data vertices of the non-VC query vertices plus a small constant. Thus, memory usage is small in practice (Section 5.2.3).

Second, we require the vertex cover U_i of q_i to be contained in the vertex cover U_{i+1} of q_{i+1}, i.e., $U_i \subseteq U_{i+1}$. This is because when we match q_{i+1}, if $U_i \not\subseteq U_{i+1}$, many groups of matches of q_i would need to be generated and processed at the same time so that the matches of q_{i+1} could be re-grouped, which incurs a high overhead.

Third, the `Extend` operator computes extensions directly in the compressed format and maintains the non-VC sets in a compressed group by directly using the intersection results, so as to avoid extra computation for compression and decompression.

The Extend algorithm. Algorithm 1 presents the `Extend` operator. Given a compressed group C_i of partial matches of a sub-query q_i, where the partial matches have the same grouping key based on a vertex cover U_i of q_i, `Extend` extends C_i to match the $(i+1)$-th query vertex u as follows. Let P be the set of *parent query vertices* of u, i.e., those query vertices that are neighbors of u but are before u in the matching order, and U_{i+1} be the selected VC of q_{i+1}. There are three cases: ❶ $P \subseteq U_{i+1}$; ❷ $P \cap U_{i+1} = \emptyset$; and ❸ $P \cap U_{i+1} \neq \emptyset$ and $P \not\subseteq U_{i+1}$.

Note that $u \in U_{i+1}$ for Cases 2 and 3, as otherwise some edge connecting u and a parent vertex in P is not covered. Denote the neighbor set of v that matches a parent query vertex p as $\mathcal{A}^u_p(v)$, in which the neighbors are candidates of the target query vertex u. As various filtering rules are applied for pruning, e.g., filtering by query vertex label, avoiding duplicate vertices in the same partial match, the neighbors of v for matching different u can be different.

In Case 1 (Lines 1-5), the *mapping set* S of u (i.e., the set of data vertices that are used to map to u) is computed by calling `ExtendFromKey`, which computes the intersection of the neighbor sets of the data vertices in C_i that are mapped to the query vertices in P. Specifically, in Line 16, `getNeighborSetIntersection`(u, P_{key}, C_i) computes $\bigcap_{p \in P_{key}} \mathcal{A}^u_p(C_i.\text{getKey}(p))$. If u is in the vertex cover U_{i+1}, then for each vertex $v \in S$, `addKey` is called to add v to C_i to generate a new group C_{i+1}, and v is added to the grouping key of C_{i+1}. Otherwise, `addSet` is called to add S to C_i to generate a new group C_{i+1}. Figure 2 shows an example, where the black vertices are the VC for each sub-query q_i. Consider extending q_1 to q_2, we have $u = u_2$ and $P = \{u_1\}$ and $U_{i+1} = U_2 = \{u_1\}$, which is Case 1 since $P \subseteq U_2$. There are two compressed groups matching (u_1): (v_0) and (v_1). To extend $C_1 = (v_0)$, we first get the mapping set of u from the neighbor set of v_0 (all neighbors whose

Algorithm 1: `Extend`$(u, C_i, P, U_i, U_{i+1})$

```
1  if P ⊆ U_{i+1} then                                              // Case 1
2      S ← ExtendFromKey (u, C_i, P)
3      if u ∈ U_{i+1} then
4          foreach v ∈ S do yield C_i.addKey(u, v)
5      else yield C_i.addSet(u, S)
6  else if P ⋂ U_{i+1} = ∅ then                                     // Case 2
7      p_min ← arg min_{p∈P} |C_i.getSet(p)|
8      S_min ← C_i.getSet(p_min)
9      S ← getNeighborSetUnion (S_min)
10     foreach v ∈ S do yield ExtendFromSet (u, v, C_i, P)
11 else                                                             // Case 3
12     P_key ← P ⋂ U_{i+1},   P_set ← P \ P_key
13     S ← ExtendFromKey (u, C_i, P_key)
14     foreach v ∈ S do yield ExtendFromSet (u, v, C_i, P_set)
15 function ExtendFromKey(u, C_i, P_key):
16     return getNeighborSetIntersection (u, P_key, C_i)
17 function ExtendFromSet(u, v, C_i, P_set):
18     C'_i ← C_i
19     foreach p ∈ P_set do
20         newset ← intersect (C'_i.getSet(p), A^u_p(v))
21         if newset = ∅ then return null
22         C'_i.updateSet(p, newset)
23     return C'_i.addKey(u, v)
```

label is not 'B' are filtered out, see v_0's neighbors in Figure 1b), which gives $S = \{v_3, v_4\}$. Since u_2 is not in U_2, we simply add S to C_1 to obtain $C_2 = (v_0, \{v_3, v_4\})$ as shown in Figure 2. Here, C_2 is a compressed group storing two partial matches (v_0, v_3) and (v_0, v_4).

There is a special scenario in Case 1 when $P \subseteq U_{i+1}$ but $P \nsubseteq U_i$, i.e., some parent query vertices mapped to the sets in C_i are added into the VC of q_{i+1}. In this case, we need to partially decompress C_i and generate multiple groups by enumerating the product of those mapping sets. Then we call `Extend` on each generated group.

In Case 2 (Lines 6-10), we first select a parent query vertex p_{min} with the smallest mapping set S_{min}. Then, we compute a *candidate set* S for u as the union of the neighbor sets of all vertices in S_{min}, i.e., $S \leftarrow \bigcup_{v \in S_{min}} \mathcal{A}^u_{p_{min}}(v)$. For each candidate $v \in S$, v is a data vertex that can be used to map to u if v is a neighbor of all the parent data vertices of u. Thus, for each $v \in S$, we call `ExtendFromSet` to compute the intersection between v's neighbor set and the mapping set of each $p \in P$. If the intersections (for all $p \in P$) are not empty, then a mapping data vertex v for u is found and a new group is generated, and v is added to the grouping key (since u is in the VC). To illustrate, consider extending q_3 to q_4 in Figure 2, we have $u = u_4$, $P = \{u_2\}$ and $U_4 = \{u_1, u_3, u_4\}$, which is Case 2 since $P \bigcap U_4 = \emptyset$. Consider to extend the compressed group $(v_0, \{v_3, v_4\}, v_2)$ of q_3, we have $p_{min} = u_2$ and $S_{min} = \{v_3, v_4\}$. We compute $S = \{v_9, v_{10}\}$ by the union of the neighbor sets of v_3 and v_4 (all neighbors whose label is not 'D' are filtered out). We then call `ExtendFromSet` to

Algorithm 2: Backtrack(q, G, U, order $\{u_1, \cdots, u_n\}$)

```
1  Let S be the set of candidates for u_1
2  if u_1 ∈ U_1 then
3      foreach v ∈ S do
4          Let C_1 ← (v); RecursiveExtend (2, C_1)
5  else Let C_1 ← (S); RecursiveExtend (2, C_1)
6  function RecursiveExtend(i, C_{i-1}):
7      if i = n + 1 then return Output (C_{i-1})
8      Let P ← {j : 1 ≤ j < i, e(i, j) ∈ q}                    // Parents
9      foreach C_i ∈ Extend(u_i, C_{i-1}, P, U_i, U_i + 1) do
10         RecursiveExtend (i + 1, C_i)
```

intersect v_9's neighbor set with u_2's mapping set, which gives $\{v_3, v_4\}$ and a new compressed group $(v_0, \{v_3, v_4\}, v_2, v_9)$ of q_4. We also intersect v_{10}'s neighbor set and u_2's mapping set, which gives $\{v_3\}$ and another group $(v_0, \{v_3\}, v_2, v_{10})$ of q_4.

In Case 3 (Lines 11-14), we divide P into two sets: P_{key} that are a part of the VC and P_{set} that are the remaining vertices in P. We compute the mapping set of u with P_{key} as in Case 1, and for each $v \in S$, generate new compressed groups with P_{set} as in Case 2.

3.3 Backtracking

Algorithm 2 shows the entire backtracking process. We begin to match a query by initializing the compressed group(s) for the first query vertex u_1 in a given matching order. If u_1 is selected in VC at the beginning, then each mapping data vertex of u_1 forms a compressed group (Line 4); otherwise, the set of mapping data vertices of u_1 forms a single compressed group (Line 5). The algorithm then recursively extends each compressed group to find matches of q by DFS, using `Extend` as a generator.

3.4 Selection of Vextex Covers

We first analyze how compression-based backtracking reduces redundant computation and then introduce a strategy for selecting the VC for each sub-query.

Consider $U_i \subseteq P$. In this case, all matches of q_i that contain the same parent data vertices of u must be in the same compressed group, and thus they can share the set intersection results. Therefore, there is no redundancy in extending the matches of q_i to those of q_{i+1} that could have been saved by any better grouping. Now consider $U_i \not\subseteq P$, the matches of q_i that contain the same parent data vertices of u may be divided into multiple groups due to different *non-parent data vertices* in the grouping keys, i.e., the data vertices mapping to $w \in U_i \setminus (U_i \cap P)$. Therefore, a VC that results in less such grouping keys is preferred for less computation redundancy.

Cost model. We devise a cost model to estimate the cost of the computation required to extend q_i to q_{i+1}:

$$\begin{aligned} cost_i(U) &= \underbrace{card_{q_i}(U) \cdot |P_{key}|}_{\text{ExtendFromKey}} + \underbrace{card_{q_i}(U \cup \{u\}) \cdot |P_{set}|}_{\text{ExtendFromSet}} \\ &= card_{q_i}(U) \cdot |U \cap P| + card_{q_i}(U \cup \{u\}) \cdot (|P \setminus (U \cap P)|), \end{aligned} \tag{1}$$

where $card_{q_i}(U)$ is the (estimated) number of compressed groups processed for extending q_i, which are generated based on a vertex cover U. The first term in the equation corresponds to the estimated total number of intersections for all `ExtendFromKey` calls, where $|P_{key}|$ is the number of intersections in one `ExtendFromKey` call. The second term corresponds to the estimated total number of intersections for all `ExtendFromSet` calls, where $|P_{set}|$ is the number of intersections in one `ExtendFromSet` call, and $card_{q_{i+1}}(U \cup \{u\})$ estimates the total size of all S in Case 2 or 3 among all C_i of q_i (i.e., the number of calls to `ExtendFromSet` when $P_{set} \neq \emptyset$). Note that $card_{q_i}(U) \leq card_{q_i}(U \cup \{u\})$ since adding u to the grouping key will only create more finer groups. Thus, a vertex cover U that is smaller and overlaps more with P is likely to give a smaller $cost_i(U)$.

To estimate $card_{q_i}(U)$, we use the product of the cardinality of the *candidates* (i.e., possible matches) of each query vertex in U (e.g., using the label frequency or degree distribution in the data graph based on the label and degree of the query vertex). Alternatively, when the matching sets of the query vertices are generated, we may compute $card_{q_i}(U)$ as the possible backtracking search space following path weight computation in [3, 14] for a tighter estimation.

Search strategy. Algorithm 3 shows how to generate the VCs. Let $\langle q_1, q_2, \ldots, q_n \rangle$ be a sequence of sub-queries defined by a matching order, where n is the number of vertices in a given query q. Even under the *containment constraint* $U_i \subseteq U_{i+1}$ (discussed in "Design choices" above), the search space to find the best VC for each q_i is large.

Instead of searching the entire space, which is $O(2^{n(n+1)/2})$ size for a combination of n levels with 2^i enumerations at each level i, we generate a relatively small search space $\mathbb{U}$ with size $O(n^2)$, as shown in Lines 1-10. We first generate a *minimum-weight vertex cover* U_i^i for each sub-query q_i by branch and bound (Line 3), using $card_{q_i}(\{u\})$ as the weight of each query vertex u in q_i. Next, for each i, we compute a minimum-weight vertex cover U_j^i for each q_j, where $1 \leq j \leq n$ and $j \neq i$, while enforcing the containment constraint $U_j^i \subseteq U_{j+1}^i$. Specifically, we use an assignment table to pre-assign the vertices according to U_i^i, and run branch and bound on the vertices with null assignment.

Then, we use dynamic programming (DP) to find the sequence of VCs that minimizes the total cost $TC = \sum_{i=1}^{n-1} cost_i(U_i)$ for matching $q = q_n$ (Lines 11-18). We define a DP table T, where $T(i, U_i^k)$ records the minimal total cost for matching q_{i+1} when U_i^k is selected as the VC for q_i. We initialize $T(1, U_1^k)$ for each U_1^k (note $U_1^k = \{u_1\}$ or $\emptyset$):

$$T(1, U_1^k) = cost_1(U_1^k) = \begin{cases} card_{q_1}(\{u_1\}), & \text{if } U_1^k = \{u_1\} \\ card_{q_1}(\{u_2\}), & \text{otherwise } U_1^k = \emptyset \end{cases}$$

Then, we recurrently compute $T(i, U_i^k)$ for $1 < i < n$ and $U_i^k \in \mathbb{U}_i$. We set $\hat{U}_{i-1}$ as the parent of U_i^k for back-tracing in the DP table (Line 17). To retrieve the best sequence of VCs, we select the vertex cover $U_{n-1} \in \mathbb{U}_{n-1}$ that minimizes $TC = T(n-1, U_{n-1})$ and trace back recursively. For the VC of q, we adopt the minimum-weight vertex cover $U_n \supseteq U_{n-1}$ (Line 19). The time complexity of DP is $O(n^2 \cdot t)$, where $t = \Omega(n)$ is the complexity of computing $card_{q_i}(U)$.

3.5 Discussion

How do we compare with BFS/DFS approaches? Our method conducts DFS at the granularity of compressed groups. Compared with existing BFS and DFS approaches, which handle partial matches individually, our method can be categorized as a hybrid BFS/DFS approach, but with a different objective and technical solution from all existing hybrid approaches. Existing works that combine BFS and DFS [7, 12, 39, 44, 46] aim at balancing between memory usage and parallelization efficiency. They use BFS to increase parallelism instead of reducing repeated computation. In fact, BFS-based approaches are widely used by GPU-based solutions [8, 12, 13] or distributed solutions [7, 38, 44]

Algorithm 3: `GenerateVertexCovers`(q, $card$)

$\mathbb{U} \leftarrow \{\emptyset : i \in [1, n]\}$
for $1 \le i \le n$ **do** `// Populate the search space`
  $U_i^i \leftarrow$ `MinimumWeightVertexCover` $(q_i, card)$
  $\mathbb{U}_i \leftarrow U_i^i \cup \mathbb{U}_i$
  foreach $1 \le j \le n, j \ne i$ **do**
    Let A be the assignment table for $u_1, \cdots, u_j$
    foreach $u \in \{u1, \cdots, u_j\}$ **do**
      Let $A(u) \leftarrow \begin{cases} 0, & \text{if } j < i \text{ and } u \notin U_i^i \\ 1, & \text{if } j > i \text{ and } u \in U_i^i \\ null, & otherwise \end{cases}$
    $U_j^i \leftarrow$ `MinimumWeightVertexCover` $(q_j, card, A)$
    $\mathbb{U}_j \leftarrow U_j^i \cup \mathbb{U}_j$
Let T be a two-dimensional cost table
foreach $U_1^k \in \mathbb{U}_1$ **do** $T(1, U_1^k) \leftarrow cost_1(U_1^k)$
for $1 < i < n$ **do**
  foreach $U_i^k \in \mathbb{U}_i$ **do**
    $\hat{U}_{i-1} \leftarrow \underset{U_{i-1}^j \in \mathbb{U}_{i-1}, U_{i-1}^j \subseteq U_i^k}{\text{argmin}} \; T(i-1, U_{i-1}^j)$
    $T(i, U_i^k) \leftarrow T(i-1, \hat{U}_{i-1}) + cost_i(U_i^k)$
    Set $\hat{U}_{i-1}$ as the parent of U_i^k
$U_{n-1} = \text{argmin}_{U_{n-1}^j \in \mathbb{U}_{n-1}} \; T(n-1, U_{n-1}^j)$
$U_n \leftarrow U_{n-1}$ if U_{n-1} is a VC of q else $U_{n-1} \cup \{u_n\}$
return a sequence with all ancestors of U_{n-1}, U_{n-1}, and U_n

to exploit massive parallelism and provide good load balancing (Section 6). From a technical point of view, existing hybrid approaches either take a batch-based approach (e.g., handling batches in a DFS manner but BFS within each batch) [7, 12, 44] or switch between BFS and DFS [39, 46] dynamically based on the number of generated partial matches. The batching or switching is either based on a preset threshold or runtime memory availability, which are query-independent. For example, PBE-REUSE [13] allocates an equal portion of the available memory to each level of the backtracking tree for storing intermediate results, and switches from BFS to DFS when the buffer at the current level is full. In contrast, our solution makes use of the query pattern to maximize computation sharing. BFS happens virtually within a compressed group (here, "virtually" means the partial matches in a group are not really enumerated), while the groups are evaluated in a DFS manner. Furthermore, our approach can be applied to single-thread solutions (e.g., CFL, GQL) to improve their performance (Section 5.2.1), while existing hybrid approaches are only for parallel scenarios.

How do we compare with worst-case optimal join? Isomorphic subgraph matching is NP-complete [9]. This problem can be formulated as a join problem in database research, where each query edge represents a relation, and each query vertex represents an attribute. Worst-case optimal join (WCOJ) guarantees that the running time is proportional to the worst-case output size, which is theoretically better than binary join. We discuss how our method relates to WCOJ.

Ngo et al. [26] propose a generic WCOJ algorithm that generalizes seminal algorithms including LeapFrog TrieJoin [41], which is adopted in a commercial database. In the context of subgraph matching, WCOJ processes all relations associated with a query vertex u in a single multiway join, for which the time complexity is proportional to the smallest relation size. WCOJ uses backtracking and processes a "query vertex at a time", as does Circinus, in contrast to binary join that processes a "query edge at a time". Circinus is an instantiation (with shared computation) of the generic WCOJ algorithm [26] when the vertex cover enables only Case 1 in Algorithm 1. Specifically, the generic WCOJ algorithm allows an arbitrary partitioning of the query vertex set for recursive query decomposition, and Circinus with only Case 1 partitions the (sub)query q_i by $\{u1, \cdots, u_{i-1}\}$ and $\{u_i\}$. `ExtendFromKey` is essentially a multiway sort-merge join that has running time proportional to the smallest neighbor set size[3]. For Cases 2 and 3, the computation for the candidate set S cannot be guaranteed to have running time proportional to the size of the smallest relation (i.e., smallest parent neighbor set). However, Circinus uses a cost model (refer to details in Section 3.4) for plan optimization and only allows Cases 2 and 3 in the plan when it is cheaper than the alternative plans with only Case 1. We compare with LeapFrog TrieJoin in Section 5.2.

Can existing solutions be easily extended to compression-based backtracking? Our compression-based method is designed for backtracking-based solutions and orthogonal to many existing techniques that optimize the matching order, candidate filtering, or data graph layout. We demonstrate in Section 5.2 that our method can be applied in existing algorithms (i.e., CFL and GQL) with advanced filtering and ordering techniques, and we further improve their performance significantly. However, from the implementation perspective, existing subgraph matching solutions cannot be easily extended to support compression-based backtracking for the following reason. The logic for subgraph matching is written in a nested for-loop in existing solutions, and both the optimization logic (such as automorphism elimination and candidate filtering) and the query evaluation logic (such as set intersection and enumeration) are tightly integrated with the data structure that stores partial matches. However, compression-based backtracking uses compressed groups to represent partial matches, which requires us to change the data structure for storing partial matches in existing solutions, and this alone requires retouching most of the code base in existing solutions. This motivates us to build a general framework. We provide a modular design with the "extend" operators for abstracting the backtracking logic, an optimizer for generating an operator chain with add-on filters and checks, and a push-based execution model. With this design, compression-based optimization and other techniques can be easily combined to process different types of queries and graphs (e.g., labeled, unlabeled) that require different techniques.

4 QUERY EXECUTION

To apply the compression-based backtracking algorithm, we propose a subgraph matching framework, called *Circinus*, as well as some optimizations for query execution.

4.1 The Circinus Framework

To enable our method as a general optimization that can be applied on existing solutions to improve their performance, we design Circinus as a general optimization and execution framework. The framework is a modular design so that various techniques for optimizing each step of subgraph matching can be easily plugged in. Figure 3 shows an overview of the framework, which consists

[3]For each target vertex u, the generic WCOJ algorithm (semi-)joins the relations corresponding to all query edges incident to u. We only intersect the neighbor sets associated with P_{key}, and implement semi-join on u's neighbors that are not yet matched by a degree or neighbor label frequency filter on the smallest neighbor set before intersection.

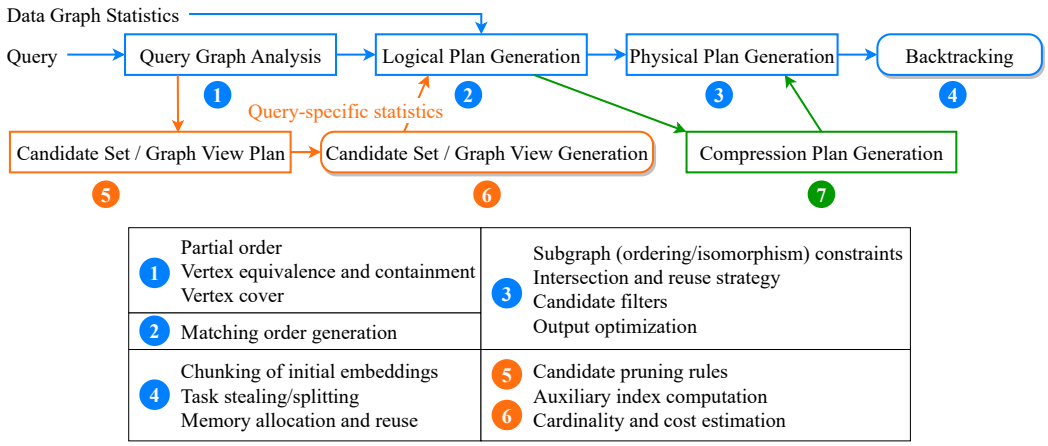


Fig. 3. A modular general subgraph matching framework

of seven modules and the flow among them. On the right we also list a summary of representative techniques used in each module.

Modules 1 to 4 are general for backtracking-based subgraph matching, and some optimization techniques in these modules are commonly used in existing solutions. Modules 1 to 3 form the query planning logic, which first analyzes a query graph without the knowledge of the data graph or the execution layer, then uses the analytical results to generate a logical plan, and finally constructs a physical plan based on the logical plan and the information from the execution layer. Module 4 then executes the physical plan by backtracking and may use parallelization for intensive computation.

Modules 5 and 6 support specialized optimizations such as various pruning rules (e.g., data vertex labels, degree, neighborhood label frequency, existence of neighbors in candidate sets, etc.) to prune candidate matches for each query vertex. Auxiliary data structures may also be built using the refined candidate sets to facilitate backtracking. Query-specific costs (e.g., estimation of subgraph counts and extension cost) are calculated from the auxiliary data structure or the data graph, which can be used to compute the costs of alternative matching orders. Module 7 generates a compression-based query plan for computation sharing (Section 3).

Figure 3 shows the general workflow in Circinus with two planning and two execution phases. The first planning phase consists of Modules 1 and 5, followed by Module 6 in the first execution phase. If no candidate set needs to be generated in Module 5, the execution in Module 6 is skipped. The second planning phase consists of Modules 2, 7 and 3, followed by the final execution by Module 4.

4.2 Compression Specific Optimizations

Circinus supports several optimizations that are specific to compression-based backtracking.

Strategic pruning of compressed groups. During backtracking, a compressed group is only fully decompressed when we output the matched subgraphs. However, some compressed groups can be found containing no valid matches due to matching the same data vertex to multiple query vertices. This happens due to two reasons. First, multiple mapping sets in the same compressed group may contain overlapping data vertices when they match the query vertices of the same label. Specifically, if there are k such mapping sets but they jointly contain less than k distinct vertices,

the whole compression group is invalid and should be pruned as early as possible. Second, when a vertex is added to a compressed group as key, it may appear in some of the existing mapping sets.

We devise a pruning strategy that applies on each compressed group when it is generated, in order to avoid useless computation on invalid groups. Consider a compressed group that matches q_i, let the label of u_i be l. We set a pruning threshold θ_l, which is the number of query vertices with label l in q_i. We extract all mapping sets that have label l and size smaller than θ_l. Then we check whether there is at least one tuple in their Cartesian product that does not have duplicates and does not overlap with a key with label l. If no tuple is found, we prune the compressed group.

Note that for a valid compressed group, overlapping vertices may exist in different mapping sets, but this does not incur useless computation because: (1) in the case of `ExtendFromKey`, the mapping sets are not involved in the computation, and the computation for the valid and invalid partial matches are shared in the group; and (2) in the case of `ExtendFromSet`, the mapping sets associated with the parent query vertices are independently updated by intersecting the neighbor vertices of the target data vertex.

Support for algorithms using auxiliary data structures. Algorithms such as GQL [16] and CFL [3] compute a candidate vertex set for each query vertex and build auxiliary bipartite graphs for processing each given query. The space complexity of building the bipartite graphs is high, which is $O(|E_G| \times |E_q|)$, where $|E_G|$ is the number of edges in data graph G and $|E_q|$ the number of edges in query q. Thus, we avoid constructing the bipartite graphs for better scalability, but add the candidate set of the target query vertex for intersection in `ExtendFromKey` and compute the candidate edges on the fly. Note that since Circinus already avoids repeated computation, it does not benefit much from using bipartite graphs (which trades space for time).

Adaptive extension from compression sets. In Case 2 of Algorithm 1, a target set S is computed by union. When an algorithm such as GQL [16] and CFL [3] is used and generates a small candidate set for the target query vertex, whose size is smaller than the estimated union size, computation can be reduced by directly using the candidate set as S instead of computing the union. The computation for S is adapted during runtime for the extension of each compressed group.

4.3 Parallelization and Load Balancing

To support efficient parallel computation, Circinus uses chunking and a lowest-level-first task splitting strategy for load balancing.

Chunking. In the case of multi-thread execution, Circinus divides the initial partial matches into chunks. A task is assigned to run an entire execution pipeline on one chunk. As the search space starting from each initial partial match can be different, Circinus makes use of the costs of the initial partial matches computed in the query planning phase to generate load-balanced chunks. In case the chosen planning strategy does not compute a cost, the degrees of the data vertices mapping to the first query vertex are used.

Lowest-level-first task splitting. As the costs used for generating chunks can be inaccurate, Circinus dynamically maintains load balance during query execution. If the number of unfinished tasks becomes smaller than the available threads, a signal is sent to the running tasks for task splitting. Instead of computing and splitting a batch of outputs at the current level of the DFS tree when a signal is received, a task first splits partial matches in its input chunk to create new tasks. When there is no partial matches to split at the current lowest level (i.e., the unfinished level that is closest to the root of the DFS tree), the task computes partial matches at the next level and split them to create new tasks. Thus, the task always computes and splits at the lowest possible level. The reason for this lowest-level-first strategy is that the partial matches at a lower level have more computation in expectation, and splitting at the lowest level first makes the original task and new tasks to search in the sub-trees that are rooted at the same level, so that they have similar loads.

Table 1. Datasets

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\Sigma\rvert$	Density
Human (HM)	4,674	86,282	44	7.90×10^{-3}
Youtube (YT)	1,134,890	2,987,624	30	4.64×10^{-6}
Youtube2 (YT2)	2,430,449	21,851,126	13	7.40×10^{-6}
LiveJournal (LJ)	3,997,962	34,681,189	40	4.34×10^{-6}
Orkut (OR)	3,072,441	117,184,899	40	2.48×10^{-5}
Friendster (FR)	65,608,366	1,806,067,135	100	8.39×10^{-7}

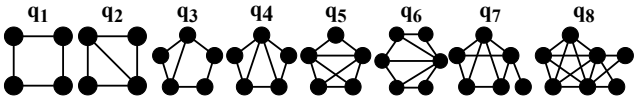


Fig. 4. Unlabeled queries

5 EXPERIMENTAL EVALUATION

We evaluated Circinus on different datasets and query settings to answer the following three questions:

(1) *How does Circinus compare with the state-of-the-art solutions?*
(2) *How effectively can Circinus reduce redundant computation?*
(3) *How do various settings in Circinus (e.g., $card_{q_i}(U)$, parallelism) affect its performance?*

5.1 Experimental Setup

Datasets. We used 6 real-world datasets that have been popularly used for the evaluation of previous subgraph matching method [3, 17, 22, 24, 36]. Table 1 lists their statistics. HM is a biological network with labels. YT2 describes Youtube videos posted from February to May in 2007, where each video is represented as a vertex, two videos have an edge if they are related, and the video category is the label. The other datasets are unlabeled social networks, for which labels are randomly assigned to data vertices from a label set following previous works [3, 14].

Queries. We used 8 unlabeled queries, as shown in Figure 4, where q_1-q_7 were also used in the evaluation of Peregrine [17], CBF [28] and PBE [12, 13] (PBE used q_1, q_2 and q_5). For labeled queries, we generated queries by extracting subgraphs from each dataset by random walk and BFS, following previous works [2, 3, 14, 15, 30, 36]. We generated 4 sets of queries, $\{Q_6, Q_8, Q_{12}, Q_{16}\}$, where each set Q_n consists of 200 queries and $n \in \{6, 8, 12, 16\}$ is the number of vertices in each query.

All experiments, unless otherwise specified, were run on machines with two 2.1GHz Intel(R) Xeon(R) Silver 4110 CPU with hyper-threading (16 cores, 32 hyper-threads) and 128GB RAM running 64-bit Debian release 9.8.

5.2 Performance Comparison

We first report the results of comparing Circinus (1) against Peregrine [17], LeapFrog TrieJoin (LFTJ) [41], CFL [3], and GQL [16] on *labeled queries*, and (2) against Peregrine, LFTJ, GraphPi [34], PBE-REUSE [13], and CBF [28] on *unlabeled queries*. CFL and GQL are two efficient algorithms for *labeled* subgraph matching. Peregrine is the state-of-the-art graph mining system that supports subgraph matching. LFTJ is a seminal worst-case optimal join algorithm adopted in the commercial database LogicBlox. PBE-REUSE and CBF are specialized *unlabeled* subgraph matching methods, where PBE-REUSE is a multi-thread system and CBF is based on MapReduce. For CFL and GQL,

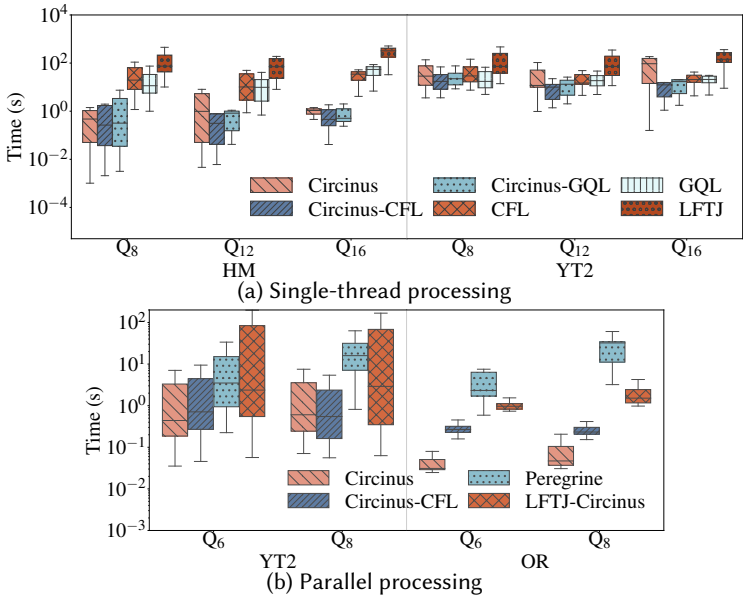


Fig. 5. Execution time for labeled queries

we used the implementation of CFL and GQLfs provided in [36] (GQLfs [36] is shown to give the best overall performance among the state-of-the-art algorithms for labeled queries). For Peregrine, PBE-REUSE and GraphPi, we obtained the source code from the authors. For PBE-REUSE, we used its multi-thread implementation. For LFTJ, we implemented the algorithm by strictly following the specification in [41] (including the trie iterator interface and join logic). As LFTJ does not specify a matching order, we use the matching order of GQL to run LFTJ because the ordering strategy of GQL is reported to be most effective among the state-of-the-art subgraph matching methods [36]. As [41] does not discuss other query optimizations (e.g., automorphism handling) or how to parallelize the single-thread algorithm, we also implemented the join logic of LFTJ on Circinus (without using trie iterators for relations, but optimized direct access to CSR) and disabled compression, and we denote this implementation as *LFTJ-Circinus*.

As LFTJ, CFL and GQL are single-thread algorithms, we compared Circinus with them by running on one thread. For Peregrine, GraphPi, CBF, and PBE-REUSE, we compared with them on a machine using all 32 hyper-threads. For CBF, we used the recommended setting of 1 core and 4 GB memory per container, except that we used 6GB memory per container for the largest dataset FR.

5.2.1 Results for Labeled Queries. For labeled queries, we compared Circinus with CFL [3] and GQL [16], which make use of candidate filtering rules and auxiliary data structures for query optimization, as well as LFTJ. *Circinus* is the solution using only general modules (i.e., Modules 1-4) and Module 7, while *Circinus-CFL* and *Circinus-GQL* further include the matching order and candidate filtering strategy of CFL and GQL (implemented in Modules 5-6).

Single-thread processing. Figure 5a reports the time of the six methods for processing query sets Q_8, Q_{12}, and Q_{16} on HM and YT2. We use the standard box plots [43] to report the time for each set of queries. The CFL/GQL based solutions have more advantage than LFTJ, especially for processing more complex queries, thanks to their advanced pruning strategies. Circinus-CFL and Circinus-GQL significantly improve the performance of CFL and GQL by compressed-group-based

Table 2. Number of completed queries

	HM			YT2		
	Q_8	Q_{12}	Q_{16}	Q_8	Q_{12}	Q_{16}
Circinus-CFL	125	78	37	125	76	55
Circinus-GQL	124	75	37	124	76	55
Circinus	99	51	15	97	44	40
LFTJ	42	23	13	87	49	39
CFL	83	32	18	114	67	49
GQL	98	34	19	125	68	52

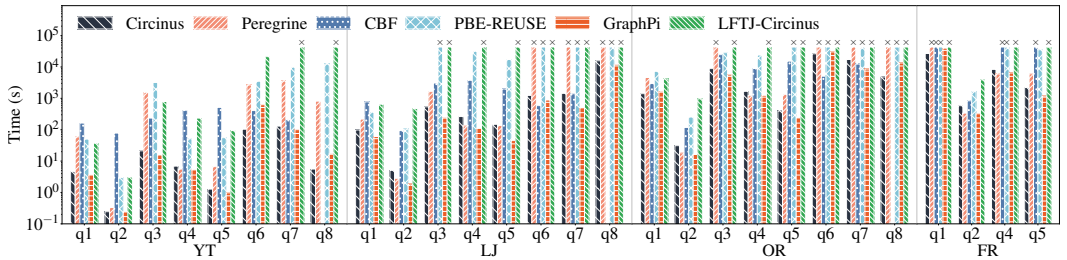


Fig. 6. Execution time for unlabeled queries

backtracking on both HM and YT2 for queries of different sizes, even though both CFL and GQL already have advanced optimizations for reducing redundant computation by matching order and candidate filtering. Comparing Circinus with CFL/GQL, we can see that computation sharing in Circinus is obviously more effective than candidate filtering in CFL/GQL for processing HM, but their performance is comparable for YT2. This is because HM is a much denser graph, which presents more opportunity for computation sharing as the same set of data vertices may appear in many partial matches.

As a single thread has insufficient computing power and may take a long time to process a query, we killed a query if it could not be completed in 1,800 seconds (we exclude queries that could not be completed by CFL, GQL, LFTJ and Circinus from Figure 5a). Table 2 reports the number of queries that were completed by each method. With the help of computation sharing, Circinus-CFL and Circinus-GQL completed considerably more queries than others.

Parallel processing. We then compared the parallel query processing performance of Circinus with Peregrine and LFTJ (implemented on Circinus). However, Peregrine cannot process larger queries within reasonable time and it also takes too long to process HM (HM is the smallest graph but is dense) and FR. Thus, we selected YT2 and OR and used query sets Q_6 and Q_8. We will report the results of Circinus on larger queries and other datasets in subsequent experiments.

Figure 5b reports the results, where all solutions used 32 threads. Both Circinus and Circinus-CFL outperform Peregrine and LFTJ-Circinus, and Circinus is up to two orders of magnitude faster than Peregrine. Both Peregrine and Circinus conduct backtracking directly on the data graph without using complex candidate filtering or auxiliary data structure. Thus, the performance improvement of Circinus over Peregrine is mainly due to computation sharing by compressed-group-based backtracking. Although Circinus does not offer the worst-case optimal guarantee as LFTJ-Circinus, Circinus significantly outperforms LFTJ-Circinus because Circinus has the advantage of computation sharing based on a practical cost model (see Discussion on worst-case optimal join in Section 3.5).

Circinus even outperforms Circinus-CFL (also Circinus-GQL) on OR, which is because the overhead of candidate generation of CFL is larger than the benefits brought by CFL. The different performance of Circinus for HM/YT2 and OR can be explained by the label distribution in the data graphs. HM and YT2 have skewed label distribution, but OR has a uniform label distribution. Thus, the complex filtering rules of CFL/GQL have barely more pruning power than the simple label/degree-based filtering of Circinus in the case of OR.

5.2.2 Results for Unlabeled Queries. Figure 6 reports the execution time for unlabeled queries. We used the same matching orders for LFTJ and Circinus. Some baseline systems can take long time to process a query, as the number of matches for an unlabeled query can be very large. Thus, we killed a query when it has taken more than 12 hours to process. For the largest graph FR, all the systems ran over 12 hours for processing q_3 and q_6-q_8, and we omit the results of those queries in Figure 6. Figure 6 also skips the results of q_8 for CBF, since CBF requires hand-optimized MapReduce implementation for the queries and we ran its built-in queries (i.e., q_1-q_7) only.

Among these systems, i.e., Circinus, GraphPi, LFTJ, PBE-REUSE, Peregrine, and CBF, there is no single winner. Overall, Circinus is significantly better than other methods in most cases except comparing with GraphPi. For small unlabeled queries, graph automorphism and query vertices with equivalent neighborhood are the dominant factor of computation redundancy. These problems are addressed in GraphPi by its 2-cycle based automorphism elimination and computation-avoid techniques. However, while GraphPi is the state-of-the-art system specialized for unlabeled subgraph matching, its techniques have limited effects for labeled subgraph matching. In contrast, Circinus has good performance for both labeled and unlabeled queries. PBE-REUSE is a cache-based approach specialized for unlabeled subgraph matching, which extends PBE [12] by generating an execution plan to exploit cached set intersection results. However, PBE-REUSE is inefficient to search the cached results during execution and it also suffers from high memory footprint. CBF has the advantage of computation reuse by massive materialization, but it suffers from high memory footprint and disk read-write overhead. While materialized triangles allows more computation reuse, the overhead can outweigh the benefit especially for processing smaller and denser queries (e.g., q_1, q_2, q_5). In contrast, Peregrine is efficient in memory usage by backtracking, but due to the common problem of a DFS approach as analyzed in Section 2, Peregrine suffers from much redundant computation. Especially for larger queries (e.g., q_6-q_8), Peregrine performs poorly as repeated computation increases quickly when the backtracking search space becomes bigger. In comparison, Circinus enjoys both benefits of memory efficiency and computation sharing brought by compressed-grouped based backtracking, and achieves the best overall performance. In most cases, its performance is from a few times to an order of magnitude better than Peregrine and CBF. Circinus consistently outperforms LFTJ-Circinus because of its compression-based computation sharing strategy.

5.2.3 Memory Usage Analysis. We report the peak memory consumption of the solutions we compared in the above experiments. The results (Figures 7 and 8) show that the memory overhead of Circinus is small (not much larger than a DFS-based approach and negligible comparing with a BFS approach).

Figure 7a reports the peak memory usage of single-thread processing for labeled queries corresponding to Figure 5a. Compared with CFL and GQL, Circinus, Circinus-CFL and Circinus-GQL have comparable peak memory usage on HM. Circinus also has comparable peak memory usage with the DFS approach LFTJ. As Circinus computes one compression group at a time, the overhead of compression-based backtracking compared with a DFS approach is bounded by the number of non-vertex-cover query vertices (i.e., the number of mapping sets in a compressed group) multiplied by the max degree in the data graph (i.e., the largest possible size of a mapping set). Thus, the

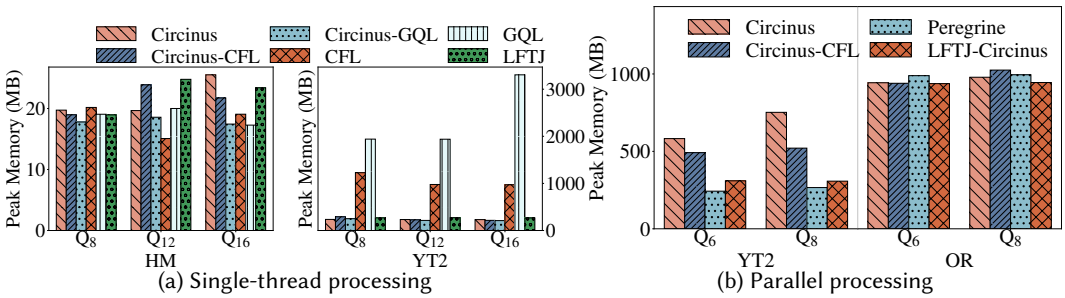


Fig. 7. Peak memory for labeled queries

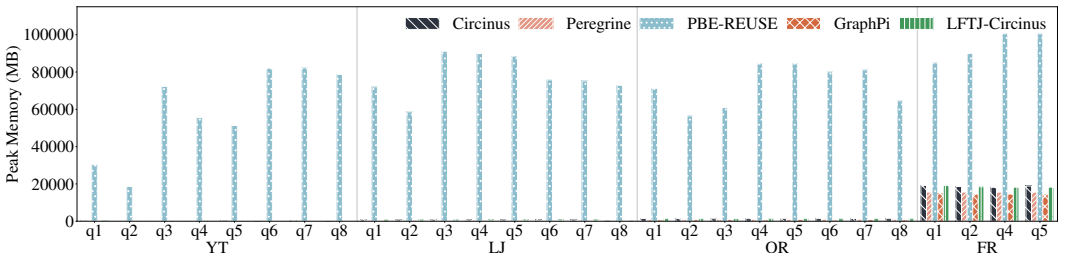
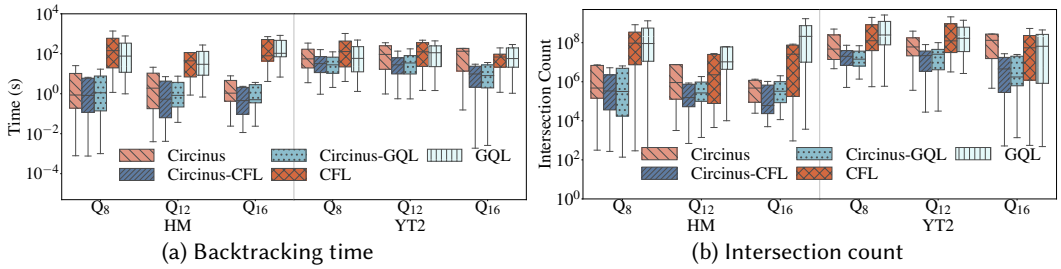


Fig. 8. Peak memory for unlabeled queries

memory overhead of compression is normally small. For YT2, Circinus has much less memory usage than CFL and GQL, because Circinus avoids constructing the auxiliary data structure (as discussed in 4.2). CFL and GQL build auxiliary structures for computation reuse and the memory for the auxiliary structures of YT2 accounts for over 90% of the memory usage because YT2 has a skewed label distribution.

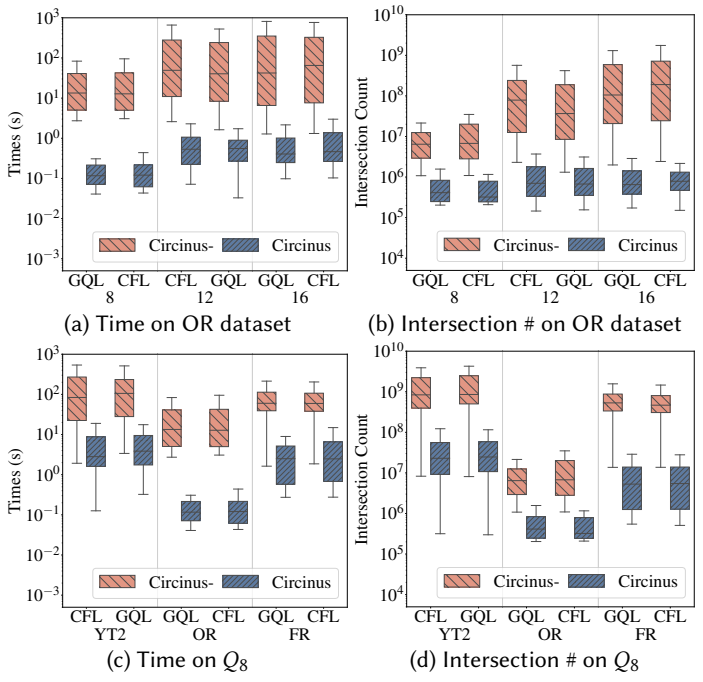
Figures 7b and 8 report the peak memory usage of parallel processing corresponding to Figures 5b and 6. From Figure 7b, the compression-based solutions (i.e., Circinus and Circinus-CFL) have more memory usage than the DFS-based solutions on YT2 (i.e., Peregrine and LFTJ-Circinus), but their memory usages are comparable on OR. Note that the extra memory usage is not solely due to compression, as Circinus and LFTJ have comparable memory usage in the single-thread case (Figure 7a), but it is also because of the parallelization mechanism in Circinus. When a thread becomes idle and tries to steal work from other threads, Circinus makes the longest running tasks generate multiple groups at a time. Since YT2 has a skewed distribution, the loads among threads are more often unbalanced. Thus, task splitting happens more often and generates many groups, leading to higher memory overhead. However, as shown in Figure 8, comparing with a mostly BFS approach (i.e., PBE-REUSE), the memory usage of DFS-based approaches and Circinus are negligible for smaller graphs (YT, LJ, OR) and also significantly smaller for the large graph FR. We do not report the memory usage for CBF in Figure 8 because it allocates all memory for containers at the beginning, and CBF has very high memory demand as well as disk I/O usage to handle the massive intermediate results in BFS. Therefore, we can conclude that the memory overhead of our compression-based method is reasonable, and Circinus achieves its goal of both memory efficiency and computation sharing.

In the following sections, we focus on labeled queries because there are only a small number of unlabeled queries whose processing time is acceptable, i.e., only certain small query graphs, while most queries are expensive to process using any existing methods.



(a) Backtracking time (b) Intersection count

Fig. 9. Backtracking time vs. intersection counts



(a) Time on OR dataset (b) Intersection # on OR dataset

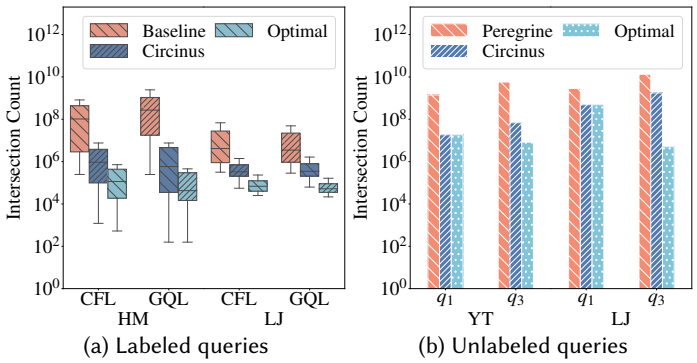
(c) Time on Q_8 (d) Intersection # on Q_8

Fig. 10. Backtracking time and intersection count with different query sizes and data graphs

5.3 Effectiveness of Redundancy Reduction

In this set of experiments, we first show that the performance gain of Circinus mainly comes from the reduction of redundant computation. Then we show for which query sizes and datasets redundancy reduction is more effective. We count the number of set intersections required in each case. The difference in the intersection counts between Circinus and the baselines reflects the saved computation.

Effect of redundancy reduction. Figure 9 reports the backtracking time (excluding the time for planning and candidate generation) and the intersection counts for Circinus, Circinus-CFL, Circinus-GQL, CFL and GQL. The pattern of the backtracking time in Figure 9a is similar to that in the intersection count in Figure 9b, suggesting that the intersection count has a direct influence on the backtracking time. Note that Circinus-CFL/GQL adopts the same candidate filtering strategy and matching order as in CFL/GQL, which leads to a similar backtracking search space. Thus,



(a) Labeled queries (b) Unlabeled queries

Fig. 11. Actual and optimal intersection #

the difference in the intersection counts mainly comes from Circinus's compressed-group-based backtracking. The results therefore verify that compressed-group-based backtracking is an effective redundancy reduction technique and it is the main factor of Circinus's performance improvement over CFL and GQL.

Degree of redundancy reduction. We also evaluated to what degree Circinus reduces repeated computation. We ran CFL and GQL for labeled queries using Q_8, and Peregrine for unlabeled queries q_1 and q_3. We report the intersection counts and the corresponding optimal intersection counts that are achievable by eliminating all repeated computation at each level of backtracking in Figure 11[4]. We have discussed the results in Section 2 to verify our observation. Although the optimal number is hard to achieve using a DFS-based approach, Circinus effectively saves an average of 96.45% redundant computation in the unlabeled case and 87.30% in the labeled case compared with the baselines.

Sensitivity to query sizes and data graphs. We disabled VC-based group compression in Circinus, denoted by *Circinus-* in Figure 10, and evaluated the effect of VC-based group compression on query performance with varying query sizes and data graphs. Candidate filtering and matching order of CFL and GQL were also enabled, as labeled by CFL and GQL in the x-axis in Figure 10. Figures 10a-10d show that Circinus significantly outperforms Circinus- for all query sizes and data graphs, thus validating the effectiveness of VC-based group compression. The results also show that the reduced redundancy (i.e., the reduction in the intersection count from Circinus to Circinus-) effectively transforms into performance gain (i.e., the shortened backtracking time) in Circinus.

Figure 10b shows that VC-based group compression can reduce more intersections for larger queries, which also results in greater reduction in the backtracking time in Figure 10a (note that the y-axis is in log scale). Figure 10d further shows that the reduction in intersections is greater for larger dataset (i.e., FR) and dataset with skewed label distribution (i.e., YT2), and Figure 10c shows that queries for these datasets are also more expensive to process and the significant reduction in the intersection count is reflected in a great magnitude of reduction in the backtracking time.

5.4 Influence of Vertex Covers

We evaluated the quality of Circinus's VC selection method by comparing with an alternative that uses the minimum-weight VC of the input query graph q for compression (denoted by *MWVC* in Figure 12). MWVC computes the VC of each sub-query q_i of q by intersecting the minimum-weight

[4]Note that the optimal counts using GQL and CFL for the same query can be different due to different matching orders and filtering strategies used.

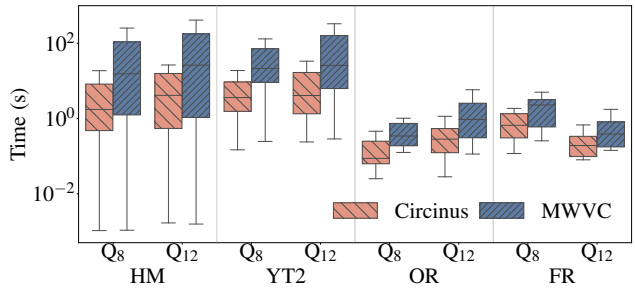


Fig. 12. Influence of VCs on performance

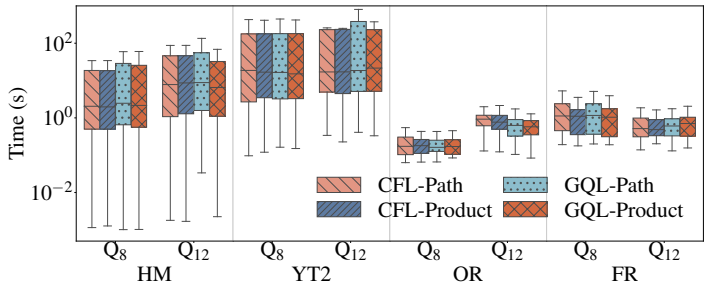


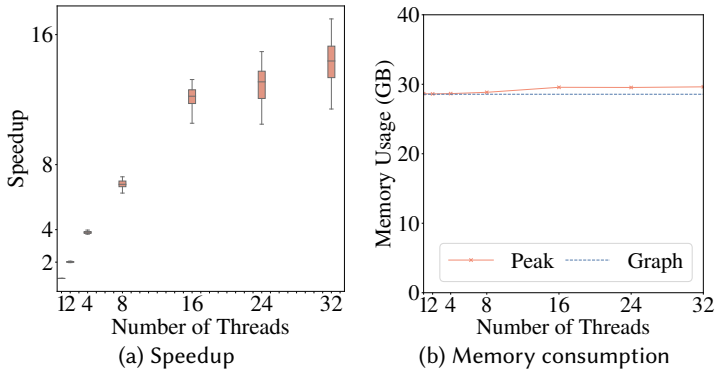
Fig. 13. Influence of $card_{q_i}(U)$ estimation on performance

VC of q with the vertex set of q_i. Figure 12 reports the backtracking time of MWVC and Circinus, both using CFL. Circinus outperforms MWVC for all datasets and queries. Note that MWVC actually optimizes the compression ratio of the outputs. In contrast, Circinus selects the VCs to aim for set intersection reduction.

5.5 Sensitivity to $card_{q_i}(U)$ Estimation

We compared two estimation methods of $card_{q_i}(U)$: (1) compute $card_{q_i}(U)$ by calculating tree-like path weights based on the auxiliary data structure constructed by CFL and GQL, denoted by *CFL-Path* and *GQL-Path*; and (2) compute $card_{q_i}(U)$ by the product of the cardinality of the candidates of the query vertices in U, denoted by *CFL-Product* and *GQL-Product*. For simplicity, here we use CFL/GQL to refer to Circinus-CFL/GQL. We want to show that a simple and efficient estimation method such as Product is also effective.

Figure 13 reports the backtracking time for running Q_8 and Q_{12} on four datasets. Although the estimation of $card_{q_i}(U)$ is more accurate using path weights than using product, a less accurate estimation is not necessarily less effective for our purpose. In fact, for most queries, the selected VC sequences are the same using either of the estimation methods. This is because that the difference in $card_{q_i}(U)$ for different U is dominated by the size of U, regardless of path weights or product being used, since most query vertices have a large number of matches. As a result, in Equation 1, $card_{q_i}(U \cup \{u\})$ is usually much larger than $card_{q_i}(U)$, and thus minimizing the cost leads to the selection of a smaller vertex cover U that has more intersection with the parent vertex set P.



(a) Speedup (b) Memory consumption

Fig. 14. Performance of parallelization

5.6 Performance of Parallelization

We evaluated the performance of Circinus by varying the number of threads from 1 to 32, for processing query set Q_{16} on the largest dataset FR. Figure 14a reports the speedup of the execution time using multiple threads over a single thread. Circinus achieves almost linear scalability up to 8 threads and then the speedup slows down from 8 threads to 16 threads. The degradation on the scalability can be explained by the hardware in our machine, which has 16 physical cores (32 hyper-threads) and 8 cores per socket. When Circinus uses more than 8 threads, more than two sockets are used and thus the performance is affected by the NUMA effect. Also, when more than 16 threads are used, the computation power does not increase as much as using more physical cores due to hyper-threading.

Figure 14b reports the peak memory consumption (labeled as *Peak*) and the memory used to store the input graph (labeled as *Graph*). With compressed-group-based backtracking, Circinus achieves small memory usage for query processing (i.e., the difference between Peak and Graph). The peak memory slowly increases when the threads are increased from 1 to 16, and remains around 29.56GB when the number of threads is further increased to 32. The result shows that Circinus has stable and low memory usage.

6 RELATED WORK

Graph mining systems. There are several graph mining systems that support labeled/unlabeled subgraph matching [7, 11, 17, 24, 38, 42, 44] (here, we exclude those graph mining systems that do not support subgraph matching). Arabesque [38] provides high-level APIs for specifying the mining process and proposes the "think like an embedding" concept. G-Miner [7], G-thinker [44] and RStream [42] adopt similar high-level abstractions as Arabesque. G-Miner proposes a task oriented processing framework to optimize memory usage and improve processing speed. G-thinker provides an intuitive graph-exploration API for users to implement graph mining algorithms and solves the high memory usage problem by buffering excess tasks in a disk-based priority queue. RStream is the first single-machine, out-of-core graph mining system with a set of relational algebra operations to express mining tasks as relational joins. As pointed out in [24], the above four systems still suffer from high memory usage and hence computation inefficiency due to their BFS-like execution approach. AutoMine [24], Fractal [11] and Peregrine [17] are the more recent graph mining systems, which adopt the DFS approach for efficient memory usage. Fractal extends "think like an embedding" to fractoids to allow intelligent planning of edge-induced, vertex-induced

or pattern-induced execution. AutoMine proposes a set-based representation to save space and provide the foundation for its compilation techniques for code generation. Peregrine provides a pattern-based programming model with anti-edge/vertex semantics. Peregrine also adopts the set-based representation as in AutoMine. Among these systems, Peregrine and Fractal make use of automorphism elimination to reduce computation redundancy for subgraph matching. We compared Circinus with Peregrine in Section 5 as it is a more recent graph mining system and has shown to outperform other systems [7, 11, 38, 42].

Subgraph matching systems and algorithms. Compared to graph mining systems, specialized subgraph matching systems provide more sophisticated optimizations. GraphZero [23] improves over AutoMine using group theory to break symmetry in the query graph for automorphism elimination. It allows orientation optimization for arbitrary patterns by requiring higher-degree vertices to have smaller IDs. GraphPi [34] improves over GraphZero by generating multiple partial orders for automorphism elimination and uses a cost model to jointly select matching order and partial order. GraphPi also uses the inclusion-exclusion principle on vertex sets to optimize subgraph counting. PBE [12] and Pangolin [8] make use of GPU to accelerate computation. Pangolin proposes a high-level extend-reduce-filter model for GPU processing and uses a BFS-based approach to exploit GPU parallelism. PBE proposes to partition a data graph and a shared execution approach to find inter-partition matches. PBE-REUSE [13] extends PBE to reuse set intersection results by a caching-based approach, and it provides both GPU and CPU multi-thread implementation. HUGE [46] and aDFS [39] switch between BFS and DFS to strike a balance between memory efficiency and parallelism. HUGE is a distributed join-based framework with a push-pull hybrid communication model. It focuses on optimizing distributed (hash and worst-case-optimal) join operations. aDFS uses asynchronous DFS-based backtracking to bound memory consumption and switches to BFS at the same backtracking level when more local work is needed for parallelism.

Many subgraph matching algorithms have been proposed. We only discuss the most related ones and refer readers to a recent survey [36]. GQL [16], Turbo$_{iso}$ [15], CFL [3], CECI [2], DAF [14] and VEQ [20] support labeled queries by following a "preprocessing-enumeration" [36] approach to prune the backtracking search space. They first generate a candidate set for each query vertex using various filtering rules and then build an auxiliary data structure that consists of bipartite graphs corresponding to query edges. Based on the auxiliary data structure, they use a heuristic to select matching orders and conduct backtracking. Turbo$_{iso}$ compresses query graphs according to vertex neighbor set equivalence to reuse computation. Boost$_{iso}$ [30] extends this idea by compressing a data graph based on neighbor set equivalence and containment to reduce computation redundancy. SEED [21] and CBF [28] support unlabeled queries following a join-based approach [1] and are implemented on MapReduce. SEED precomputes a set of non-overlapped cliques in a data graph for compression in order to avoid materialization of partial matches. CBF proposes a VC-based compression technique to reduce the cost of outputting and storing subgraph matches, and it computes matches by joining edges and cliques.

Compared with existing subgraph matching systems and algorithms, Circinus addresses an intrinsic problem of repeated computation in backtracking-based subgraph matching [2, 3, 11, 12, 14–17, 23, 24, 34, 39, 41, 46] and we have shown in Section 5 that our method significantly outperforms the state-of-the-art methods [3, 12, 16, 17, 28, 36]. Circinus is also a general framework that allows flexible combinations of many optimization techniques to optimize the processing of different datasets and query sets.

Use of compression and vertex cover. Boost$_{iso}$ [30] and SEED [21] use data graph compression to reduce computation redundancy, but they have heavy preprocessing costs and the benefit requires special local structures (e.g., many vertices with similar neighbors or large cliques). Circinus reduces

computation redundancy without preprocessing as it compresses partial matches on-the-fly during backtracking, and thus can be used to benefit dynamic graph processing. Peregrine [17] uses VC to reduce isomorphism checking overhead, CBF [28] compresses the output subgraphs using a minimum VC to save disk write and storage costs, and PBE [12] uses VC to delay materialization of partial matches. Their use of VC is different from that in Circinus, which uses VC to group partial matches for computation sharing and achieves better performance than Peregrine, CBF and PBE as reported in Section 5.2.

Graph database systems. Graph databases [6, 10, 29, 37] support different semantics for isomorphic or homomorphic subgraph matching. They adopt the property graph model and support query languages such as Cypher and Gremlin. Different from the aforementioned isomorphic subgraph matching solutions whose workloads are characterized by complex query topology and intensive whole-graph access, the property graph databases are designed for workloads that are characterized by selective property filters and simple query topology (e.g., the popular LDBC benchmark). Therefore, the two sides employ very different techniques. Specifically, the performance of a system for processing LDBC-like workloads depends largely on the index design and graph layout of the graph databases. For example, Neo4J [37] supports b-tree indexes on a single or multiple properties to accelerate vertex/edge retrieval. RedisGraph [29] supports single-property indexes. It partitions the adjacency matrix of a graph by edge labels and represents each partition in CSR in memory to allow fast linear-algebra-based traversal on edges with specific labels. TigerGraph [10] applies homomorphic encoding on property graphs for compact storage, which allows fast single vertex/edge retrieval and index update. Grasper [6] partitions a property graph in a distributed cluster and optimizes access to the graph topology and properties by a native RDMA-based graph storage design. In conclusion, while isomorphic subgraph matching solutions and graph databases work on the same problem at the core, they are designed to handle queries of different characteristics.

7 CONCLUSIONS

We presented Circinus - a subgraph matching system that provides a new compression-based method to reduce computation redundancy in backtracking. The proposed method can be generally applied in combination with the optimization techniques in existing solutions such as CFL, GQL, and Peregrine. Our experimental results demonstrate that Circinus significantly reduces repeated computation while maintaining low memory footprint, which transfers to up to several orders of magnitude better performance than the state-of-the-art solutions.

REFERENCES

[1] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704. https://doi.org/10.14778/3184470.3184473

[2] 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, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1447–1462. https://doi.org/10.1145/3299869.3300086

[3] 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* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1199–1214. https://doi.org/10.1145/2882903.2915236

[4] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* 14, S-7 (2013), S13. https://doi.org/10.1186/1471-2105-14-S7-S13

[5] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-Performance Dynamic Pattern Matching over Disordered Streams. *Proc. VLDB Endow.* 3, 1 (2010), 220–231. https://doi.org/10.14778/1920841.1920873

[6] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 87–100. https://doi.org/10.1145/3357223.3362715

[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, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 32:1–32:12. https://doi.org/10.1145/3190508.3190545

[8] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[9] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman (Eds.). ACM, 151–158. https://doi.org/10.1145/800157.805047

[10] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[11] Vinícius Vitor dos Santos Dias, Carlos H. C. Teixeira, Dorgival O. Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1357–1374. https://doi.org/10.1145/3299869.3319875

[12] 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 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1067–1082. https://doi.org/10.1145/3318464.3389699

[13] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Transactions on Knowledge and Data Engineering* (2020), 1–1. https://doi.org/10.1109/TKDE.2020.3035564

[14] 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* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1429–1446. https://doi.org/10.1145/3299869.3319880

[15] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 337–348. https://doi.org/10.1145/2463676.2465300

[16] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 405–418. https://doi.org/10.1145/1376616.1376660

[17] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 13:1–13:16. https://doi.org/10.1145/3342195.3387548

[18] Alpár Jüttner and Péter Madarasi. 2018. VF2++ - An improved subgraph isomorphism algorithm. *Discret. Appl. Math.* 242 (2018), 69–81. https://doi.org/10.1016/j.dam.2018.02.018

[19] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1695–1698. https://doi.org/10.1145/3035918.3056445

[20] 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 *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 925–937. https://doi.org/10.1145/3448016.3457265

[21] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228. https://doi.org/10.14778/3021924.3021937

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[23] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 21–37. https://doi.org/10.1145/3469379.3469383

[24] Daniel Mawhirter and Bo Wu. 2019. AutoMine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 509–523. https://doi.org/10.1145/3341301.3359633

[25] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[26] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. https://doi.org/10.1145/2590989.2590991

[27] N. Pržulj, D. G. Corneil, and I. Jurisica. 2006. Efficient Estimation of Graphlet Frequency Distributions in Protein–Protein Interaction Networks. *Bioinformatics* 22, 8 (April 2006), 974–980. https://doi.org/10.1093/bioinformatics/btl030

[28] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: On Compression and Computation. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 176–188. https://doi.org/10.14778/3149193.3149198

[29] RedisLabs. 2021. RedisGraph - a graph database module for Redis. https://oss.redislabs.com/redisgraph/

[30] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 617–628. https://doi.org/10.14778/2735479.2735493

[31] Carlos R. Rivero and Hasan M. Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowl. Inf. Syst.* 51, 1 (2017), 61–87. https://doi.org/10.1007/s10115-016-0968-2

[32] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431. https://doi.org/10.1145/3186728.3164139

[33] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 364–375. https://doi.org/10.14778/1453856.1453899

[34] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: high performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 100. https://doi.org/10.1109/SC41405.2020.00104

[35] Tom A. B. Snijders, Philippa E. Pattison, Garry L. Robins, and Mark S. Handcock. 2006. New Specifications for Exponential Random Graph Models. *Sociological Methodology* 36, 1 (2006), 99–153. https://doi.org/10.1111/j.1467-9531.2006.00176.x

[36] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1083–1098. https://doi.org/10.1145/3318464.3380581

[37] The Neo4J Team. 2021. Neo4J. https://neo4j.com/

[38] Carlos H. C. 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, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 425–440. https://doi.org/10.1145/2815400.2815410

[39] Vasileios Trigonakis, Jean-Pierre Lozi, Tomás Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 209–224. https://www.usenix.org/conference/atc21/presentation/trigonakis

[40] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42. https://doi.org/10.1145/321921.321925

[41] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR* abs/1210.0481 (2012). arXiv:1210.0481 http://arxiv.org/abs/1210.0481

[42] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 763–782. https://www.usenix.org/conference/osdi18/presentation/wang

[43] Wikipedia contributors. 2021. Box plot — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Box_plot&oldid=1059408900 [Online; accessed 11-December-2021].

[44] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, Wei-Shinn Ku, and John C. S. Lui. 2020. G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1369–1380. https://doi.org/10.1109/ICDE48307.2020.00122

[45] 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* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/1007568.1007607

[46] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2049–2062. https://doi.org/10.1145/3448016.3457237

[47] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1 (2010), 340–351. https://doi.org/10.14778/1920841.1920887

Received April 2022; revised July 2022; accepted August 2022