GPU-Accelerated Subgraph Enumeration on Partitioned Graphs

Wentian Guo, Yuchen Li*, Mo Sha, Bingsheng He, Xiaokui Xiao, Kian-Lee Tan
National University of Singapore, *Singapore Management University
{wentian,sham,hebs,tankl}@comp.nus.edu.sg
xkxiao@nus.edu.sg,*yuchenli@smu.edu.sg

ABSTRACT

Subgraph enumeration is important for many applications such as network motif discovery and community detection. Recent works utilize graphics processing units (GPUs) to parallelize subgraph enumeration, but they can only handle graphs that fit into the GPU memory. In this paper, we propose a new approach for GPU-accelerated subgraph enumeration that can efficiently scale to large graphs beyond the GPU memory. Our approach divides the graph into partitions, each of which fits into the GPU memory. The GPU processes one partition at a time and searches the matched subgraphs of a given pattern (i.e., instances) within the partition as in the small graph. The key challenge is on enumerating the instances across different partitions, because this search would enumerate considerably redundant subgraphs and cause the expensive data transfer cost via the PCI-e bus. Therefore, we propose a novel shared execution approach to eliminate the redundant subgraph searches and correctly generate all the instances across different partitions. The experimental evaluation shows that our approach can scale to large graphs and achieve significantly better performance than the existing single-machine solutions.

ACM Reference Format:
Wentian Guo, Yuchen Li*, Mo Sha, Bingsheng He, Xiaokui Xiao, Kian-Lee Tan . 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389699

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389699

1 INTRODUCTION

Given a data graph G and a pattern graph P, subgraph enumeration is to find all subgraphs in G that are isomorphic to P (i.e., *instances* of P). Subgraph enumeration is important in various domains, such as network motif discovery [12, 27], community detection [17, 40], and frequent subgraph mining [25, 38]. Thus, there have been a lot of studies on improving the performance of subgraph enumerations (such as [8, 23, 26, 34]). Since the sequential solutions for subgraph enumeration are prohibitively slow [2, 8, 14, 24, 34], many works adopt the distributed and parallel solutions (either on the cluster [22, 23, 29, 35] or on the graphics processing units (GPUs) [26, 37]). The recent works [26, 37] parallelize subgraph enumeration on GPUs and achieve significant speedups. However, these works assume the sizes of data graphs cannot exceed the GPU memory, which greatly restricts the application scope.

To overcome the limit of GPU memory, a direct solution is to store the data graph in main memory and extend the existing approaches [8, 23, 26, 29, 34, 37] to enumerate subgraphs by loading the graph into GPUs on demand. Since the data access to main memory is via a rather slow PCI-e bus between the *host* (CPUs) and *device* (GPUs), this direct solution can incur excessive communication overheads over PCI-e and hence lead to poor performance.

In this work, we propose *partition based enumeration* (**PBE**), a new approach for GPU-accelerated subgraph enumeration. We store the data graph in main memory in the form of graph partitions, each of which fits into the GPU memory. The GPU processes one partition at a time and searches all the instances within the partition as in the case when the data graph fits into the GPU memory. Therefore, the instances within each partition can be enumerated efficiently without accessing the data graph in main memory.

With the efficient intra-partition subgraph enumeration, the key challenge is searching the instances across different partitions, because this search would enumerate considerably redundant subgraphs and cause the expensive data transfer cost via the PCI-e bus. To prune the search space of finding the instances across different partitions, we need to enumerate cross-partition subgraphs to map the vertices

of the pattern graph in *multiple* match orders (Section 3.2). However, the same subgraph can be simultaneously mapped to different match orders. If we follow the existing works [8, 23, 26, 29, 34, 37] to enumerate subgraphs for one match order at a time, a large number of cross-partition subgraphs would be repeatedly enumerated for multiple match orders. As searching each cross-partition subgraph has to access the data graph in main memory, this redundant enumeration can significantly increase the PCI-e communication and thus degenerates the performance.

To reduce the communication overhead of inter-partition subgraph enumerations, we propose a novel *shared execution* approach that can group multiple match orders together and enumerate subgraphs for one group at a time. In this way, the enumerated subgraphs are shared for a group of match orders and thereby the redundant enumeration is avoided.

For an efficient and effective shared execution, we have addressed the following two challenges. First, selecting optimal match orders that minimize the execution cost is challenging. We prove that this problem is NP-hard, and devise a heuristic method that can not only efficiently generate the match orders but also achieve effectiveness to optimize the execution cost. Our theoretical analysis shows that the execution cost achieved by our heuristic method is bounded by the optimal cost multiplied by a small factor. The second challenge for shared execution is a sound algorithm to faithfully enumerate the desired instances. While most existing works [23, 26, 29, 37] use one match order to search all instances, our shared execution employs multiple match orders to enumerate only the instances across different partitions. Thus, we develop an algorithm that is provably correct to generate all the instances across different partitions.

Hereby, we summarize the contributions as follows.

- We introduce a new approach for GPU subgraph enumeration that can efficiently scale to large data graphs beyond the GPU memory.
- We propose a shared execution approach to search the instances that cross different partitions. The shared execution can avoid the redundant searches among multiple match orders.
- The experimental results show that our approach on a single machine can scale to large graphs and achieve significantly better performance than the existing single-machine (CPUs and GPUs) baselines.

The remaining part of this paper is organized as follows. Section 2 introduces the preliminaries and related works for subgraph enumeration. Section 3 provides an overview of our approach. Section 4 presents the design of shared execution. The implementation of inter-partition search is illustrated in Section 5. Section 6 reports the experimental results. Finally, Section 7 concludes this work.

2 PRELIMINARIES AND RELATED WORK

2.1 Definitions and Notations

A data graph G is defined to be an undirected, unlabeled, and connected graph. A pattern graph P is also undirected, unlabeled and connected. For a graph g, the vertices and edges of g are denoted as $V(g)$ and $E(g)$. We call the vertices $V(G)$ and edges $E(G)$ in the data graph as the *data vertices* and *data edges* respectively. Correspondingly, we have *pattern vertices* $V(P)$ and *pattern edges* $E(P)$ for the pattern graph. Normally, we use u and v to denote the pattern vertex and data vertex respectively. For a vertex $v \in V(G)$ ($u \in V(P)$), the adjacent list of v is denoted as $N(v)$ ($N(u)$). Given G and P, subgraph enumeration is defined as follows.

Definition 1 (Subgraph Enumeration). *A graph g is **isomorphic** to a pattern graph P, if there exists a bijective mapping $f : V(P) \rightarrow V(g)$, such that $(u_1, u_2) \in E(P)$ if and only if $(f(u_1), f(u_2)) \in E(g)$. Given the data graph G and pattern graph P, subgraph enumeration finds all subgraphs of G that are isomorphic to P.*

Definition 2 (Search sequence). *Given the pattern graph P, the search sequence π is a permutation of the pattern vertices $V(P)$ that reflects the order in which $V(P)$ are matched. $\pi(i)$ is the i-th vertex in π. When $1 \leq i \leq j \leq |V(P)|$, $\pi[i : j]$ denotes the set of vertices $\{\pi(k) | i \leq k \leq j\}$. Given π, if $\pi(i) = u$, the position of u on π is $\pi^{-1}(u) = i$.*

To enumerate all isomorphic mappings, which we call *instances*, we would follow a search sequence to iteratively match each pattern vertex. Before all pattern vertices are matched, we maintain an isomorphic mapping f from *some* pattern vertices to the data vertices. Such a mapping is called the *partial instance*. To formally define the partial instance, we introduce the induced subgraph. A graph g_1 is an *induced subgraph* of g_2 on the vertex set $U \subset V(g_2)$ if g_1 has the vertex set $V(g_1) = U$ and the edge set $E(g_1)$ such that $\forall v_1, v_2 \in U$, if $(v_1, v_2) \in E(g_2)$, then $(v_1, v_2) \in E(g_1)$. We denote such an induced subgraph g_1 as $g_2(U)$.

Definition 3 (Instance). *An instance of P is an* isomorphic mapping $f : V(P) \rightarrow V(g)$, *where g is a subgraph of G. The set of instances of P is denoted as $R(P)$.*

Definition 4 (Partial instance). *Given P and π, the induced subgraph of P on $\pi[1 : i]$ is denoted as P_i^π, where $1 \leq i \leq |V(P)|$. The instances of P_i^π are the partial instances of P, which are denoted as $R(P_i^\pi)$.*

Definition 5 (Graph partition). *A* partition plan Φ *of the data graph G is a division of $V(G)$ into n disjoint vertex sets. A partition G_i of the data graph G is an induced subgraph of G on the i-th vertex set with $1 \leq i \leq n$. The partition id $\rho(v)$ of a data vertex v is the label of G_i that v belongs to, i.e., $\rho(v) = i$.*

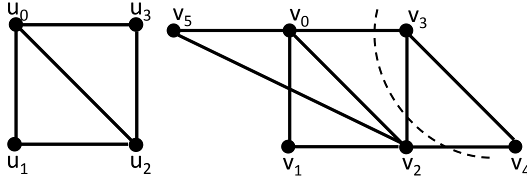


Figure 1: An example pattern graph P (left) and data graph G (right).

Given a graph partition plan Φ, an *inter-partition instance* f is an instance such that there exists $u_i, u_j \in V(P), \rho(f(u_i)) \neq \rho(f(u_j))$. An *intra-partition instance* f of P is an instance such that $\forall u_i, u_j \in V(P), \rho(f(u_i)) = \rho(f(u_j))$. We list the frequently used notations throughput the paper in Table 1.

Example 1. *Figure 1 shows an example data graph and pattern graph. Given the search sequence (u_0, u_1, u_2, u_3), we may generate the partial instances $\{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$ and $\{(u_0, v_2), (u_1, v_0), (u_2, v_3)\}$. We may find the instances $f_1 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\}$ and $f_2 = \{(u_0, v_2), (u_1, v_0), (u_2, v_3), (u_3, v_4)\}$. A partition plan on G is indicated with the dashed line. This plan divides $V(G)$ into two vertex sets, i.e., $\{v_3, v_4\}$ and $\{v_0, v_1, v_2, v_5\}$. Thus, both f_1 and f_2 are inter-partition instances.*

2.2 CPU-based Subgraph Enumerations

There are two lines of works for subgraph enumeration on CPUs, i.e., single-machine and distributed solutions.

Single-machine subgraph enumeration. Built upon the Ullman's work [39], the existing works [2, 8, 14, 24, 30, 34, 41] optimize the enumeration process with a selective search sequence [8, 14, 30, 34] and various pruning techniques [2, 8, 12, 14, 15, 30, 46, 47] to reduce the search space. The pruning techniques can be categorized into two types, namely non-indexed and indexed approaches. The non-indexed approaches can apply a series of feasibility rules, e.g., the one-look-ahead rule in VF2 [8], to invalidate unpromising partial instances beforehand. The indexed approaches mostly rely on the vertex labels to build the indexes that can prune partial instances as early as possible [2, 14, 15, 30, 46, 47]. However, the indexed approaches may not help, because subgraph enumeration focuses on unlabeled data graphs and the maintenance overheads of index structures are usually large [2, 14, 46, 47].

Distributed subgraph enumeration. To accelerate subgraph enumeration, many works rely on distributed frameworks to achieve high parallelism [22, 23, 29, 35]. While the distributed approaches seek for an efficient *scale-out* framework, this paper focuses on a *scale-up* solution that exploits the GPU power of a single machine. Thus, the distributed approaches can potentially take advantage of our solution to boost the performances.

Table 1: Frequently used notations.

Symbol	Descriptions
P, G	pattern graph and data graph
$V(g), E(g)$	vertex set and edge set of the graph g
$N(v)$	adjacent list of v
π	search sequence
P_i^π	induced subgraph of P on the vertex set $\pi[1:i]$
f	isomorphic mapping
$\rho(v)$	partition id of data vertex v
$R(P), R^r(P)$	instances and inter-partition instances of P
$Q_r = (S, H)$	inter-partition query plan
$S = \{\pi_i\}$	the set of all inter-partition search sequences
$H = \{H_l\}$	the set of prefix-equivalence groups at each level l
$[\pi]$	prefix-equivalence group with π as the representative
H^s	the set of strict-equivalence groups

2.3 GPU-based Subgraph Enumeration

Recent works have seen interests in exploiting emerging new hardware to accelerate data processing [4, 5, 42–45], especially for graph processing on GPUs [13, 32, 33]. There are two major previous studies on GPU subgraph enumeration, i.e., NEMO [26] and GPSM [37]. They are both based on a breadth-first search (BFS) approach, which is illustrated in Algorithm 1. It proceeds by a level-by-level expansion of partial instances to find all instances. At each level $l > 0$, it executes two procedures, namely Compute and Materialize. Compute iterates each partial instance $f \in R(P_{l-1}^\pi)$ to compute a candidate set $C(u|f)$ of data vertices that can match the pattern vertex u given f. To compute $C(u|f)$, we first obtain a set of neighbors of u that are matched before u (Line 6). We call such a set $N_+(u)$ the backward neighbors of u. After that, given each partial instance $f \in R(P_{l-1}^\pi)$, for the data vertices mapped to the backward neighbors $N_+(u)$, we perform a set intersection operation over their adjacent lists (Line 8). The result is a set $C(u|f)$ containing possible candidate data vertices. If any data vertex $v \in C(u|f)$ has been mapped in f, v is removed from $C(u|f)$ (Line 9). With the candidate set generated by Compute, Materialize extends each partial instance $f \in P_{l-1}^\pi$ to match one more pattern vertex and generate the new partial instance $(f \cup (u, v))$ (Line 14). This new partial instance is added to $R(P_l^\pi)$, which is a set of partial instances matched to P_l^π.

Algorithm 1 SubgEnum

Input: the pattern graph P, data graph G and search sequence π
Output: the instances $R(P)$

1: $R(P_1^\pi) = \{(\pi(1), v) | v \in V(G)\}$
2: **for** $2 \leq l \leq |V(P)|$
3: $\quad C = \text{Compute}(\pi(l), \pi, R(P_{l-1}^\pi))$
4: $\quad R(P_l^\pi) = \text{Materialize}\ (\pi(l), \pi, R(P_{l-1}^\pi), C)$

5: **procedure** $\text{Compute}(u, \pi, R(P_{l-1}^\pi))$
6: $\quad N_+(u) = \{u_i | u_i \in N(u) \wedge \pi^{-1}(u_i) < \pi^{-1}(u)\}$
7: $\quad$ **for** $f \in R(P_{l-1}^\pi)$
8: $\quad\quad C(u|f) = \cap_{\forall u_i \in N_+(u), v=f(u_i)} N(v)$
9: $\quad\quad C(u|f) = C(u|f) - \{f(u_i) | u_i \in f\}$
10: $\quad$ **Return** C
11: **procedure** $\text{Materialize}(u, \pi, R(P_{l-1}^\pi), C)$
12: $\quad$ **for** $f \in R(P_{l-1}^\pi)$
13: $\quad\quad$ **for** $v \in C(u|f)$
14: $\quad\quad\quad$ Add $(f \cup (u, v))$ to $R(P_l^\pi)$
15: $\quad$ **Return** $R(P_l^\pi)$

Although NEMO and GPSM exhibit significant performance improvements, they can only handle relatively small data graphs that are stored in the device memory. In this work, we seek to overcome this limitation and achieve efficient GPU-accelerated subgraph enumeration.

2.4 Other Related Works

General graph processing systems. Graph partitioning is widely used in general graph processing systems [6, 7, 9–11, 21], especially in distributed (e.g. PowerGraph [11]) and disk-based (e.g., GraphChi [21]) settings. These systems propose efficient execution methodology to reduce the cross-partition data accesses, but they are designed for iterative graph processing applications, such as PageRank and BFS, which maintain a small set of aggregated values. They are not suitable for subgraph enumeration that can generate a huge amount of intermediate result, which is significantly (orders of magnitude) larger than the data graph itself.

Incremental subgraph matching. One way to view the inter-partition search is to consider the inter-partition data edges as a batch of newly inserted edges and then perform incremental subgraph matching [1, 3, 18, 28]. To find the newly generated instances, the delta subgraph framework [1, 3, 18] issues one delta query Q_e for each pattern edge e. Denote the data graph before and after the edge insertions as G' and G respectively. The query Q_e would map the pattern edge e to the inserted data edges. For the other pattern edges, it would match a subset of these edges over G' and the remaining over G. Note that the pattern edges used for G' and G differ in each delta query. Thus, the intermediate results generated for different delta queries are unable to be shared. Our inter-partition search can exploit the sharing because we would only match the pattern edges over G (Section 4). This approach may cause duplicate instances, but they can be easily avoided in execution (Section 5).

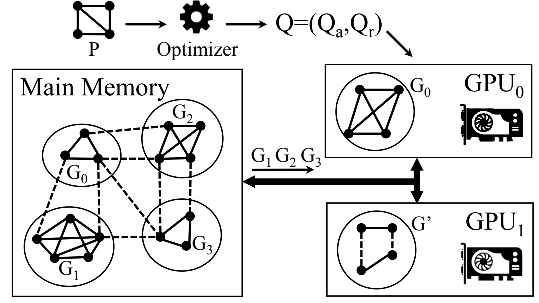


Figure 2: The overview of PBE.

3 PARTITION BASED ENUMERATION

In this section, we first give an overview of PBE and then illustrate the designs and issues for the inter-partition search.

3.1 Overview

PBE preprocesses the data graph by dividing the graph into multiple partitions, each of which can fit into the device memory. After that, the query processing workflow will enumerate subgraphs for the given pattern graph.

Query processing. Figure 2 shows the query processing workflow of PBE. Given the pattern graph P, the optimizer first generates the query plan $Q = (Q_a, Q_r)$, which consists of the plans for the intra-partition search Q_a and the inter-partition search Q_r. For the intra-partition search, the plan Q_a is one search sequence generated by following existing works [14, 26, 34, 36]. We enumerate all orders of pattern vertices and choose the order for Q_a that minimizes the execution cost. To search the intra-partition instances, we load each graph partition G_i into GPUs. After that, the intra-partition instances in G_i are enumerated using Algorithm 1 without any host-device communication.

For the inter-partition search, the plan Q_r is made up of multiple search sequences and auxiliary data structures to facilitate shared execution (Section 4.1). To search the inter-partition instances, we apply the two-stage approach (Section 3.2) that prunes all intra-partition instances by enumerating all data edges that cross different partitions, i.e., the dashed lines indicated in Figure 2. This can initialize the inter-partition partial instances, which are matched to

multiple search sequences. To avoid the redundant subgraph searches among multiple search sequences, our shared execution can group multiple search sequences and enumerate cross-partition subgraphs for one group at a time (Section 4). The details of inter-partition search are presented in Algorithm 3. After both intra-partition and inter-partition search terminate, the query processing workflow is completed.

Graph partitioning. To optimize the performance of query processing, more workload should be offloaded to the intra-partition search, since it can be executed efficiently without the host-device communication. For this purpose, we use the standard graph partition approach, i.e., METIS [19], which seeks to minimize the number of inter-partition edges. With METIS, the vertices are likely to connect to other vertices in the same partition and there would be more intra-partition instances. We partition the data graph such that each partition together with the intermediate result (which would be mentioned in Section 5) can fit into the GPU. Given the data graph size $|G|$, GPU memory size g, and the memory reserved for intermediate result θ, we set the number of partitions to be $\lceil |G|/(g-\theta) \rceil$. Note that the graph partitioning is only executed in the preprocessing step, so it would not affect the efficiency of query processing workflow.

3.2 Inter-partition Search

To enumerate all inter-partition instances, a straightforward method is to adopt the existing approach (Algorithm 1), but it can deteriorate to enumerating all instances including the intra-partition ones. The reason is that only after all pattern vertices are matched, i.e., when all the instances are found, we can determine whether the enumerated partial instance is inter-partition. Therefore, we are motivated for a new design for the inter-partition search.

Two-stage approach. We proposes a two-stage approach, which consists of the *pruning stage* and *enumeration stage*, to efficiently enumerate inter-partition instances.

Pruning stage. In this stage, we enumerate all inter-partition data edges (v_i, v_j) and map them to each pattern edge $(u_k, u_w) \in E(P)$. This can prune all intra-partition instances and initialize the partial instances that cross different partitions. For each generated partial instance, we call the first inter-partition data edge (v_i, v_j) and pattern edge (u_k, u_w) *prime data edge* and *prime pattern edge*, respectively.

Enumeration stage. In this stage, we extend the partial instances to generate inter-partition instances by following multiple search sequences. Multiple search sequences are used because the partial instances from the pruning stage have matched different pattern edges. For each pattern edge (u_k, u_w), the partial instances are extended by following a specific search sequence to match the remaining pattern vertices $V(P) - \{u_k, u_w\}$. Since we have $|E(P)|$ pattern edges, we would use $|E(P)|$ search sequences to generate the partial instances. To distinguish with the search sequence adopted in the intra-partition search, we call the search sequences used here the *inter-partition search sequences*, which are denoted as $S = \{\pi_i | 1 \leq i \leq |E(P)|\}$.

Redundant subgraph searches. The naïve design of the enumeration stage is to invoke Algorithm 1 for each search sequence separately and extend the partial instances to match all pattern vertices. However, for different search sequences, the same cross-partition subgraphs could be repeatedly generated to match the pattern vertices. As generating each cross-partition subgraph has to access the data graph in main memory, this redundant subgraph searches would excessively increase the PCI communication and thus lead to poor performance. We illustrate the redundant subgraph searches with the following example and introduce shared execution in Section 4 to address this problem.

Example 2. *Figure 3b showcases the redundant subgraph searches. Using (v_2, v_3) as the prime data edge, the generated inter-partition instances for each search sequence are depicted in the figure. Take π_1 as an example. At the beginning of the enumeration stage, we have the partial instance $f_1 = \{(u_0, v_2), (u_1, v_3)\}$. To match the pattern vertex $\pi_1(3) = u_2$, given f_1, we compute the candidate data vertices $C(u_3|f_1) = \{v_0, v_4\}$ (Lines 8-9 in Algorithm 1). This extends f_1 to generate two new partial instances $f_2 = \{(u_0, v_2), (u_1, v_3), (u_2, v_0)\}$ and $f_3 = \{(u_0, v_2), (u_1, v_3), (u_2, v_4)\}$. To match the pattern vertex $\pi_1(4) = u_3$, given f_2, we compute the candidate data vertices $C(u_3|f_2) = \{v_1, v_5\}$. This extends f_2 to generate two inter-partition instances $f_4 = \{(u_0, v_2), (u_1, v_3), (u_2, v_0), (u_3, v_1)\}$ and $f_5 = \{(u_0, v_2), (u_1, v_3), (u_2, v_0), (u_3, v_5)\}$.*

The dashed rectangles in the figure indicate the redundant subgraph searches. For the set of search sequences $\{\pi_{1-5}\}$, the partial instances matched to the first three pattern vertices have the same data vertices in the mapping (the green rectangles). For the set of search sequences $\{\pi_{1-4}\}$, the partial instances matched to the first four pattern vertices have the same data vertices in the mapping (the green and blue rectangles).

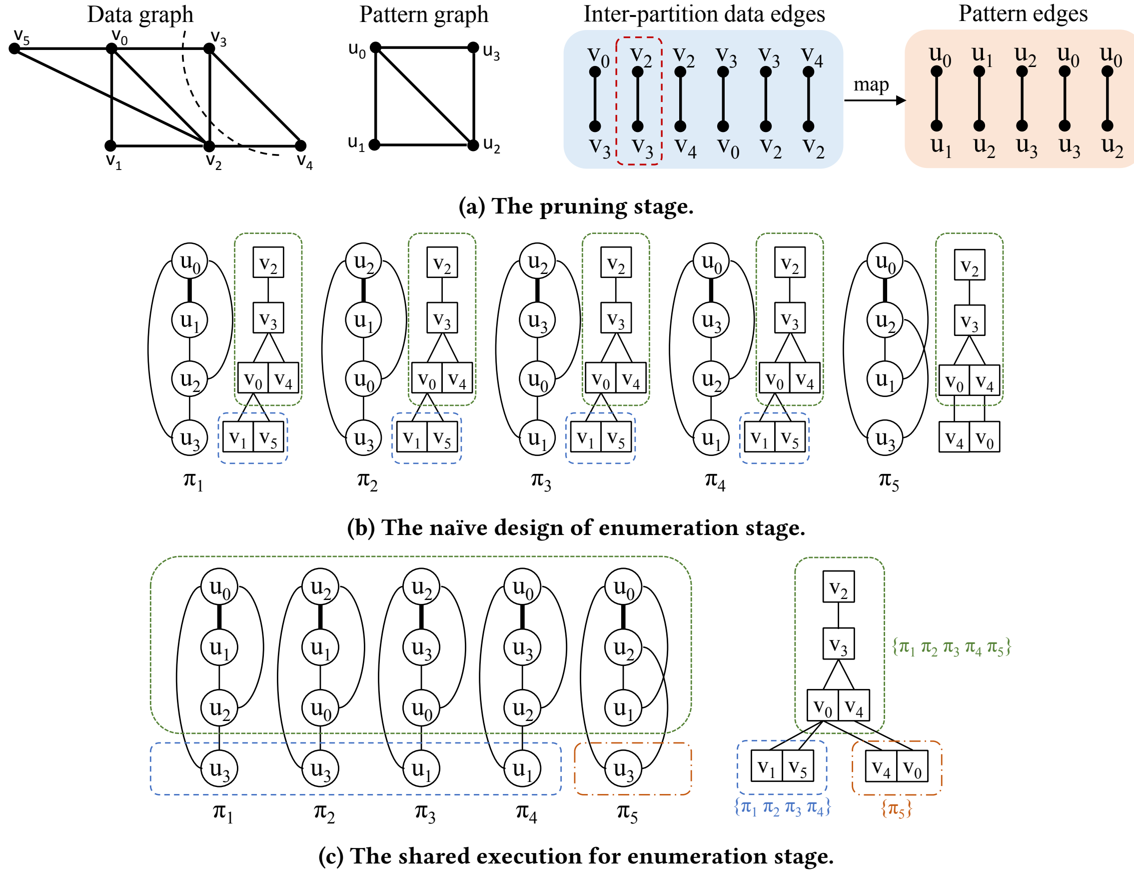
4 SHARED EXECUTION

In this section, we present the design of shared execution. As the optimal query plan is difficult to be generated, we propose a heuristic method to generate the plan.

4.1 Design

For inter-partition search, the enumeration stage can cause redundant subgraph searches among different search sequences (Example 2). This happens when the search sequences have the following property.

Definition 6 (Prefix-equivalence). *Given the search sequence π and the pattern graph P, two search sequences π_1*



(a) The pruning stage.

(b) The naïve design of enumeration stage.

(c) The shared execution for enumeration stage.

Figure 3: An example of two-stage approach for inter-partition search using the example graphs in Figure 1. In the enumeration stage, the partial instances are initialized by the prime data edge (v_2, v_3). The bold edges in the search sequences denote the prime pattern edges.

and π_2 are prefix-equivalent at level l if the induced subgraphs $P_l^{\pi_1}$ and $P_l^{\pi_2}$ are isomorphic with the mapping f that has $\forall 1 \le j \le l, f(\pi_1(j)) = \pi_2(j)$.

With the prefix-equivalence at level l as an equivalence relation, we can partition all inter-partition instances S into a set of equivalence classes H_l. Each $[\pi] \in H_l$ is a set of prefix-equivalent search sequences. We call $[\pi]$ the *prefix-equivalence group*, and π is the representative search sequence in this group. The set of all prefix-equivalence groups at each level is denoted as $H = \{H_l | 1 \le l \le |V(P)|\}$.

Example 3. *For the search sequences in Figure 3c, at levels 1, 2 and 3, all search sequences are in the same prefix-equivalence group. Thus, $H_1 = H_2 = H_3 = \{\{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5\}\}$. At level 4, there are two prefix-equivalence groups, i.e., $\{\pi_1, \pi_2, \pi_3, \pi_4\}$ and $\{\pi_5\}$, so $H_4 = \{\{\pi_1, \pi_2, \pi_3, \pi_4\}, \{\pi_5\}\}$.*

Given the search sequence π and the partial instance f that matches the vertex set $\pi[1 : l]$, we call the ordered sequence of data vertices $(f(\pi(1)), f(\pi(2)) \cdots f(\pi(l)))$ the *data sequence* of f. When two search sequences π_1 and π_2 are prefix-equivalent at level l, since $P_l^{\pi_1}$ and $P_l^{\pi_2}$ are isomorphic, the partial instances for π_1 and π_2 actually share the same data sequences. To exploit this sharing, we propose *shared execution* that can maintain common data sequences for each prefix-equivalence group.

Main idea. For a prefix-equivalence group $[\pi] \in H_l$ at level l, shared execution computes the data sequences by generating new partial instances for one representative search sequence π. The partial instances F_l for π are generated in the same way as Algorithm 1. The data sequences of F_l can also match other search sequences $\pi' \in [\pi]$ in the same group. To obtain the partial instance for π', we can directly remap the pattern vertices in F_l. Specifically, given $f \in F_l$, we generate the new partial instance for π' by $f' = \{(\pi'(1), f(\pi(1))) \cdots (\pi'(l), f(\pi(l)))\}$. This remapping can be performed in a very efficient manner in the implementation (Section 5).

Inter-partition query plan. We define the inter-partition query plan $Q_r = (S, H)$ that consists of all inter-partition

search sequences S and the prefix-equivalence groups at all levels H. Given S, H is generated accordingly. Thus, the generation of Q_r mainly needs to select the order for the search sequences S. For any search sequence $\pi \in S$, suppose the corresponding prime pattern edge is (u_k, u_w). Then $\pi[1:2]$ is either (u_k, u_w) or (u_w, u_k). For the remaining pattern vertices $\pi[3:|V(P)|]$, they could be any permutation on the set of pattern vertices $V(P) - \{u_k, u_w\}$.

Example 4. *Figure 3c shows an example of shared execution. Using (v_2, v_3) as the prime data edge, after the pruning stage, we have the data sequence (v_2, v_3) that can be mapped to all search sequences, since they are prefix-equivalent at level 2. To generate the data sequences for level 3, note that all search sequences are prefix-equivalent. We choose any one search sequence, say π_1. Given the partial instance $f_1 = \{(u_0, v_2), (u_1, v_3)\}$ matched to $\pi_1[1:2]$, we compute the candidate data vertices $C(u_2|f) = \{v_0, v_4\}$ for the pattern vertex $\pi_1(3) = u_2$. This can generate two new partial instances $f_2 = \{(u_0, v_2), (u_1, v_3), (u_2, v_0)\}$ and $f_3 = \{(u_0, v_2), (u_1, v_3), (u_2, v_4)\}$. The corresponding data sequences can be mapped to other search sequences in the same prefix-equivalent group π_{2-5}. Given f_2, to generate the new partial instance for π_2, we remap the pattern vertices in π_2 to the data sequence and have the partial instance $f_4 = \{(u_2, v_2), (u_1, v_3), (u_0, v_0)\}$. At level 4, we have two prefix-equivalent groups so the partial instances are computed for them separately. For $\{\pi_{1-4}\}$, there are two data sequences $\{(v_2, v_3, v_0, v_1)\}$ and $\{(v_2, v_3, v_0, v_5)\}$; for $\{\pi_5\}$, two data sequences are generated, i.e., $\{(v_2, v_3, v_0, v_4)\}$ and $\{(v_2, v_3, v_4, v_0)\}$*

4.2 Hardness of Order Selection

To optimize the performance of shared execution, it is crucial to select the orders of pattern vertices for inter-partition search sequences. For example, in Figure 3c, if π_2 is set to (u_1, u_2, u_0, u_3), then π_1 and π_2 are not prefix-equivalent at level 4 and the data sequences cannot be shared.

Cost model. The cost evaluation of shared execution relies on the execution cost for one search sequence. Given the search sequence π, we consider the cost $Cost(\pi, l)$ of generating inter-partition partial instances at level l.

$$Cost(\pi, l) = \begin{cases} |E^r(G)| & l \leq 2 \\ |R^r(P^{\pi}_{l-1})| \cdot K(\pi, l)) & l > 2 \end{cases} \quad (1)$$

In the pruning stage, i.e., $l \leq 2$, the overhead is to materialize the inter-partition data edges $E^r(G)$. In the enumeration stage, i.e., $l > 2$, the cost is linear to the number of inter-partition partial instances $R^r(P^{\pi}_{l-1})$ achieved at level $l-1$. Note that the superscript r is used in $R^r(P^{\pi}_{l-1})$ to indicate the partial instances here are inter-partition. We use $K(\pi, l)$ to denote the average execution cost for one partial instance $f \in R^r(P^{\pi}_{l-1})$, which includes the overhead of Compute and Materialize for f (Algorithm 1). The order of π makes a difference for $Cost(\pi, l)$ by affecting the number of partial instances $|R^r(P^{\pi}_{l-1})|$.

Objective function. Given the inter-partition query plan Q_r, we seek to minimize the execution cost $Cost(Q_r)$ of shared execution, which is defined as follows.

$$Cost(Q_r) = \sum_{l=1}^{|V(P)|} \sum_{[\pi] \in H_l} Cost(\pi, l) \quad (2)$$

For each prefix-equivalence group $[\pi] \in H_l$, shared execution only needs to compute the data sequences for only one search sequence π, and the computed data sequences can be shared for the entire group $[\pi]$.

$Cost(Q_r)$ reflects two optimization criteria for the order selection. The first criterion is to *maximize the pruning power of each single search sequence.* This decreases the number of partial instances generated and thus reduces the enumeration cost $Cost(\pi, l)$ for each search sequence π. The second criterion is to *maximize the extent of sharing among multiple search sequences.* This reduces the number of prefix-equivalence groups $|H_l|$, which can minimize the overall cost $Cost(Q_r)$. These criteria are different from the previous studies [26, 37] that choose the order for one search sequence, since they only optimize the first objective.

Hardness. Selecting an optimal order is difficult. Lemma 1 proves that this problem is NP-hard. Lemma 2 shows that the cost of exhaustive enumeration is as high as $(2 \cdot (|V(P)| - 2)!)^{|E(P)|}$, making the optimal order intractable.

Lemma 1. *Selecting an optimal order is NP-hard.*

Proof. Consider a restricted variant of the problem that optimizes only the extent of sharing: Assuming $Cost(\pi, l) = c$ is a constant, the objective function becomes $Cost(Q_r) = c \cdot \sum_{l=1}^{|V(P)|} |H_l|$. To optimize $Cost(Q_r)$, we need to find the maximum common subgraph [16], which is NP-hard. Since the restricted variant of the problem is NP-hard, so is the order selection for shared execution. □

Lemma 2. *The number of all possible combinations of inter-partition search sequences is* $(2 \cdot (|V(P)| - 2)!)^{|E(P)|}$.

Proof. For each search sequence the number of possibilities is $2 \cdot (|V(P)| - 2)!$ and we have $|E(P)|$ search sequences. □

4.3 Heuristic Search

Finding the optimal order for inter-partition search sequences is difficult, which is mainly because the prefix-equivalence exposes a large search space to enumerate all combinations of different search sequences. To restrict the search space, we propose a heuristic search method for the order selection that explores a different extent of sharing called strict-equivalence. Since strict-equivalence captures the sharing

that is more constrained than prefix-equivalence, it limits the search space and makes the order selection efficient. Meanwhile, it can effectively optimize the runtime performance for shared execution (Section 4.4).

Definition 7 (Strict-equivalence). *Given the pattern graph P and two search sequences π_1 and π_2, π_1 and π_2 are* strict-equivalent *if they are prefix-equivalent at level $|V(P)|$.*

Note the strict-equivalence is the special case of the prefix-equivalence, i.e., two search sequences are prefix-equivalent at each level. With the strict-equivalence relation, we partition the search sequences S into a set of equivalence classes H^s. Each $[\pi] \in H^s$ is called *strict-equivalence group*. We use the superscript s to differentiate H^s from the equivalence classes H partitioned by the prefix-equivalence relation. In the following, we denote the inter-partition query plan generated by the heuristic search as $Q_r = (S, H^s)$ and we may drop the superscript s when the context is clear.

Equivalence among pattern edges. We rely on the automorphism [12] to find the strict-equivalence groups. The automorphism of P is an isomorphism from P to itself, which exists when the graph is symmetric. Two pattern vertices, u_i and u_j, are considered "equivalent" if there exists an automorphism A such that $A(u_i) = u_j \vee A(u_j) = u_i$. We define the equivalence relation among pattern edges: two different pattern edges, (u_i, u_j) and (u_k, u_w), are equivalent if there exists an automorphism A such that either (1) $A(u_i) = u_k \wedge A(u_j) = u_w$ or (2) $A(u_j) = u_k \wedge A(u_i) = u_w$. This equivalence is denoted as $(u_i, u_j) \equiv (u_k, u_w)$. With this equivalence relation, we can partition all pattern edges into a number of *equivalence edge classes*. Each class consists of the equivalent pattern edges.

Finding strict-equivalent search sequences. The equivalence among pattern edges can help generate the strict-equivalent search sequences. Lemma 3 suggests that if two search sequences π_1 and π_2 have the equivalent prime pattern edges, whatever order we select for π_1, there exists an order for π_2 to be strict-equivalent to π_1. This lemma also indicates that the maximum number of strict-equivalence groups for P is the number of equivalence edge classes of P, since the search sequences can be strict-equivalent only when their prime pattern edges are equivalent.

Lemma 3. *Given the search sequence π_1 whose order is decided, the search sequence π_2 can be selected to be strict-equivalent to π_1 if and only if their prime pattern edges are equivalent.*

Proof. When the prime pattern edges are equivalent, there exists an automorphism A to map the vertices in π_1 to a sequence of pattern vertices. This sequence, which we select for π_2, can be proved to be strict-equivalent to π_1. □

Algorithm 2 Generate inter-partition query plan.

Input: pattern graph P
Output: the inter-partition query plan $Q_r = (S, H^s)$.

```
1:  EC = GenEquivalenceEdgeClass(P)
2:  used = ∅
3:  for (u_k, u_w) ∈ E(P) do
4:      if ∃(u_i, u_j) ∈ used s.t. (u_i, u_j) ≡ (u_k, u_w) then
5:          let π′ be the search sequence for (u_i, u_j)
6:          π = GenStrictEquivalenceOrder(π′, u_k, u_w)
7:      else
8:          π = GenSearchSequence(P, u_k, u_w)
9:      Add π to S
10:     Add (u_k, u_w) to used
11: H^s = GenStrictEquivalenceGroup(S)
```

Generating inter-partition query plan. Algorithm 2 illustrates our heuristic search method to build the query plan. At the beginning, the equivalence edge classes EC of P are generated (Line 1). GenEquivalenceEdgeClass finds all automorphisms [12] and then groups the equivalent pattern edges. Then we select the order for each search sequence with a different prime pattern edge (u_k, u_w) (Lines 3-10). For each pattern edge (u_k, u_w), we find a pattern edge (u_i, u_j) that (1) has been processed and (2) is equivalent to (u_k, u_w) (Line 4). The equivalence between two pattern edges is checked by verifying whether they are in the same equivalence edge class in EC. Then the search sequence for (u_k, u_w) is selected based on whether (u_i, u_j) exists.

Case 1: (u_i, u_j) exists. In this case, we would select an order for the search sequence π to make it strict-equivalent to an existing search sequence. Let π' be the search sequence with (u_i, u_j) as the prime pattern edge. GenStrictEquivalenceOrder selects an order for π that is (1) strict-equivalent to π' and (2) adopts (u_k, u_w) as the prime pattern edge (Line 6). GenStrictEquivalenceOrder can be implemented by enumerating all permutations of pattern vertices and then examining whether the enumerated order is satisfactory. Such an order is guaranteed to be found because of Lemma 3.

Case 2: (u_i, u_j) does not exist. GenSearchSequence selects an order that can maximize the pruning power and uses (u_k, u_w) as the prime pattern edge (Line 8). To maximize the pruning power, GenSearchSequence enumerates all permutations of pattern vertices and then evaluates the execution cost $Cost(\pi)$ for each given order of π.

$$Cost(\pi) = \sum_{l=1}^{|V(P)|} Cost(\pi, l) \quad (3)$$

4.4 Theoretical Analysis

We analyze the efficiency and effectiveness of our heuristic search method.

Efficiency of plan generation. For Algorithm 2, because both GenStrictEquivalenceOrder and GenSearchSequence are executed by enumerating all permutations of pattern vertices, the time complexity of each loop at Lines 4-10 is $O(|V(P)|!)$. Considering there are $|E(P)|$ pattern edges, the main procedure of Algorithm 2 at Lines 3-10 costs $O(|E(P)| \cdot |V(P)|!)$. This complexity is much smaller than finding the optimal order to explore the prefix-equivalence (Lemma 2).

Effectiveness of optimization. Let Q_r^o be the optimal query plan generated by exhaustively enumerating all orders for the search sequences. Let Q_r^s be the query plan generated by our heuristic search, i.e., Algorithm 2. We introduce Lemma 4 to compare $Cost(Q_r^o)$ and $Cost(Q_r^s)$. If the cost at the last level $Cost(\pi, |V(P)|)$ takes up more than a portion c of the enumeration cost $Cost(\pi)$ for each search sequence π, Lemma 4 proves that the ratio between the costs $Cost(Q_r^o)/Cost(Q_r^s)$ is larger than c. By Equation 1, $Cost(\pi, l)$ is linear to the number of partial instances $|R^r(P_{l-1}^{\pi})|$. Since in practice the number of partial instances at each level is always much larger than that at the previous level, the cost at the last level $Cost(\pi, |V(P)|)$ becomes the bottleneck and takes up a large portion c of $Cost(\pi)$. Since c is expected to be large, $Cost(Q_r^o)$ is close to $Cost(Q_r^s)$.

Lemma 4. *For any search sequence π, assume $Cost(\pi, |V(P)|) \geq c \cdot Cost(\pi)$, where $0 < c < 1$. Then, $Cost(Q_r^o)/Cost(Q_r^s) \geq c$.*

Proof. Let $n = |V(P)|$. For $Q_r^o = (S^o, H^o)$, we use the superscript o to denote the search sequences and prefix-equivalence groups that are generated by the optimal solution. For $Q_r^s = (S^s, H^s)$, we use the superscript s to differentiate the plan generated by our heuristic search method from Q_r^o. Let π_i^s use the same prime pattern edge as π_i^o that has the same subscript i.

By Algorithm 2, for each strict-equivalence group $[\pi] \in H^s$, there is one search sequence $\pi' \in [\pi]$ generated by GenSearchSequence to minimize the cost $Cost(\pi')$, while the remaining search sequences in $[\pi]$ are set to be strict-equivalent to π'. Thus, the order of any search sequence $\pi_j \in [\pi]$ can minimize $Cost(\pi_j)$. Then for each $\pi_i^s \in S^s$,

$$Cost(\pi_i^s) \leq Cost(\pi_i^o) \quad (4)$$

Algorithm 2 makes the search sequences with the equivalent prime pattern edges in the same strict-equivalence group. By Lemma 3, each group has maximized the set of search sequences that can be strict-equivalent. Thus, for Q_r^s, the sharing of search sequences at the last level is optimal and better than Q_r^o (the details are omitted here).

$$\sum_{[\pi] \in H_n^o} Cost(\pi) \geq \sum_{[\pi] \in H^s} Cost(\pi) \quad (5)$$

$Cost(Q_r^o)$ can be derived as follows. The first inequality is deduced by keeping only the cost at the last level $Cost(\pi_i^o, n)$. The second inequality follows by our assumption $Cost(\pi_i^o, n) \geq c \cdot Cost(\pi_i^o)$. The third inequality goes by Equation 4. The forth inequality is derived by Equation 5.

$$\begin{aligned} Cost(Q_r^o) &= \sum_{l=1}^{n} \sum_{[\pi^o] \in H_l^o} Cost(\pi^o, l) \geq \sum_{[\pi^o] \in H_n^o} Cost(\pi^o, n) \\ &\geq \sum_{[\pi_i^o] \in H_n^o} (c \cdot Cost(\pi_i^o)) \geq c \cdot \sum_{[\pi_i^o] \in H_n^o} Cost(\pi_i^s) \\ &\geq c \cdot \sum_{[\pi^s] \in H^s} Cost(\pi^s) = c \cdot Cost(Q_r^s) \end{aligned}$$

Therefore, this lemma is concluded. □

5 EFFICIENT ENUMERATION

In this section, we present the implementation of shared execution for the inter-partition search.

Algorithm. Algorithm 3 illustrates the implementation of enumerating inter-partition instances. To enumerate the inter-partition instances for a strict-equivalence group $[\pi] \in H^s$, the execution consists of three steps.

Step 1: Pruning stage. To prune all intra-partition instances, we enumerate all inter-partition data edges (v_i, v_j) and map (v_i, v_j) to the prime pattern edge $(\pi(1), \pi(2))$ (Lines 3-5). As both v_i and v_j are possible to be mapped to $\pi(1)$ or $\pi(2)$, we map (v_i, v_j) and (v_j, v_i) to $(\pi(1), \pi(2))$.

Step 2: Enumeration stage. Suppose the pattern vertex at level l is u. Let $N_+(u)$ be the backward neighbors of u, i.e., those neighbors of u that are matched before u in the search sequence π. Before Compute, LoadSubgraphToGPU would load a subgraph G' of G from main memory to the GPU (Line 9). G' consists of the adjacent lists required for the candidate set computation in Compute. To collect the required adjacent lists N_l, for each partial instance f in the currently achieved partial instances B and each backward neighbor $u_i \in N_+(u)$, the adjacent list of the data vertex $v = f(u_i)$ is added into N_l (Line 16). The set of adjacent lists N_l is fetched from the data graph G stored in main memory and transferred into the GPU (Line 17). To increase the throughput of data transfer, the adjacent lists are organized into a continuous array and then transmitted in batch.

After G' is ready, the Compute procedure in Algorithm 1 is invoked to compute the candidate set C (Line 10). When C is obtained, we will execute DupRemove to filter the elements in C that may cause duplicate inter-partition instances (Line 11). We leave the discussion on DupRemove later. With the new candidate set C_r, the materialize procedure in Algorithm 1 is called to generate the new partial instances B_l for level l (Line 12).

Algorithm 3 InterPartitionSearch

Input: the pattern graph P, data graph G, partition plan Φ, and inter-partition query plan $Q_r = \{S, H^s\}$. $S = \{\pi_i\}$ are all inter-partition search sequences and H^s is a set of strict-equivalence groups.

Output: the inter-partition instances $R^r(P)$

```
1:  for [π] ∈ H^s do
2:      ∀1 ≤ l ≤ |V(P)|, B_l = ∅                          ▷ pruning stage
3:      for (v_i, v_j) ∈ E(G) s.t. ρ(v_i) ≠ ρ(v_j) do
4:          B_2 = B_2 ∪ {(π(1), v_i), (π(2), v_j)}
5:          B_2 = B_2 ∪ {(π(1), v_j), (π(2), v_i)}
6:      for 3 ≤ l ≤ |V(P)| do                          ▷ enumeration stage
7:          u = π(l)
8:          N_+(u) = {u_i | u_i ∈ N(u) ∧ π^{-1}(u_i) < π^{-1}(u)}
9:          LoadSubgraphToGPU(B_{l-1}, N_+(u))
10:         C = SubgEnum.Compute(u, π, B_{l-1})
11:         C_r = DupRemove(B_{l-1}, C, u, N_+(u))
12:         B_l = SubgEnum.Materialize(u, π, B_{l-1}, C_r)
13:     for π_j ∈ [π] s.t. π_j ≠ π do                  ▷ exploit sharing
14:         R^r(P) = R^r(P) ∪ ReMap(B_{|V(P)|}, π, π_j)

15: procedure LoadSubgraphToGPU(B, N_+(u))
16:     N_l = {N(v) | f ∈ B, u_i ∈ N_+(u), v = f(u_i)}
17:     Load N_l from main memory
18: procedure DupRemove(B, C, u, N_+(u))
19:     for f ∈ B do
20:         e_1 = OrderPair(f(π(1)), f(π(2)))
21:         C_r(u|f) = ∅
22:         for v ∈ C(u|f), flag = true do
23:             for u_i ∈ N_+(u) do
24:                 if ρ(v) ≠ ρ(f(u_i)) then
25:                     e_2 = OrderPair(v, f(u_i))
26:                     if e_1 > e_2 then
27:                         flag = false
28:             if flag then
29:                 C_r(u|f) = C_r(u|f) ∪ v
30:     Return C_r
31: procedure ReMap(B, π_i, π_j)
32:     B' = ∅
33:     for f ∈ B do
34:         f' = {(π_j(l), v) | 1 ≤ l ≤ |V(P)|, v = f(π_i(l))}
35:         B' = B' ∪ f'
36:     Return B'
```

Step 3: Exploit sharing. With the inter-partition instances $B_{|V(P)|}$ for the search sequence π, the final step is to apply ReMap to generate the instances for any other search sequence $\pi_j \in [\pi]$ by utilizing $B_{|V(P)|}$ (Line 14). Specifically, for each instance $f \in B_{|V(P)|}$ and each search sequence $\pi_j \in [\pi]$, we map the sequence of pattern vertices $(\pi_j(1), \pi_j(2) \cdots \pi_j(|V(P)|))$ to the data sequence of f (Line 34). Note that for a search sequence $\pi_j \in [\pi]$, the used sequence of pattern vertices is the same. Thus, in our implementation, we simply materialize the sequence of pattern vertices once and map it to the data sequences of all instances $B_{|V(P)|}$. In this way, we significantly reduce the overhead of ReMap.

Avoid duplicates. Compared with Algorithm 1, Algorithm 3 introduces one additional procedure, i.e., DupRemove, to prevent generating duplicate inter-partition instances. Duplicates occur when there are multiple inter-partition data edges in the partial instance, as shown in the example below.

Example 5. *Suppose we follow the search sequences in Figure 3c to search the inter-partition instances on the example graph in Figure 1. Without duplicate removal, if starting from the prime data edge (v_3, v_0) and following π_1, (v_3, v_0, v_2, v_4) is found to match $\pi_1 = (u_0, u_1, u_2, u_3)$; if starting from the prime data edge (v_3, v_2) and following π_5, (v_3, v_2, v_0, v_4) is found to match $\pi_5 = (u_0, u_2, u_1, u_3)$. However, both mappings are the same and thus duplicates occur.*

To avoid duplicate instances, we can assign a total order for data edges and enforce the prime data edge to be the smallest in terms of the edge order. Lines 18-30 illustrate the implementation of DupRemove. For each partial instance f, e_1 is the prime data edge (Line 20). For any data vertex $v \in C(u|f)$ and any data vertex $f(u_i)$ matched to the backward neighbor u_i, we ensure v and $f(u_i)$ cannot form an inter-partition data edge smaller than e_1 (Lines 24-27).

Correctness. We introduce Lemma 5 to prove the correctness of Algorithm 3.

Lemma 5. *Algorithm 3 can generate exactly the inter-partition instances.*

Proof. Let $R^r(P)$ be the inter-partition instances generated by Algorithm 3. Denote $R^r_o(P)$ as the true inter-partition instances. We need to prove that $R^r(P) = R^r_o(P)$.

First, as each instance generated by Algorithm 3 crosses different partitions, by definition $R^r(P) \subseteq R^r_o(P)$. Then, we would show that $R^r_o(P) \subseteq R^r(P)$. For each instance $f \in R^r_o(P)$, there must exist a pair of pattern vertices $(u_k, u_w) \in E(P)$ such that the corresponding data edge $(f(u_k), f(u_w))$ is the smallest among the inter-partition data edges in f. Suppose the search sequence π is the one that has (u_k, u_w) as the prime pattern edge. Then f is generated for π in Algorithm 3 and thus $f \in R^r(P)$. Therefore, $R^r_o(P) \subseteq R^r(P)$. With $R^r(P) \subseteq R^r_o(P)$ and $R^r_o(P) \subseteq R^r(P)$, we have $R^r(P) = R^r_o(P)$. ◻

Memory management. For memory management of partial instances, we follow the previous work [26] to enumerate subgraphs in a pipeline manner. Given the memory capacity θ reserved for storing the intermediate result, we keep an

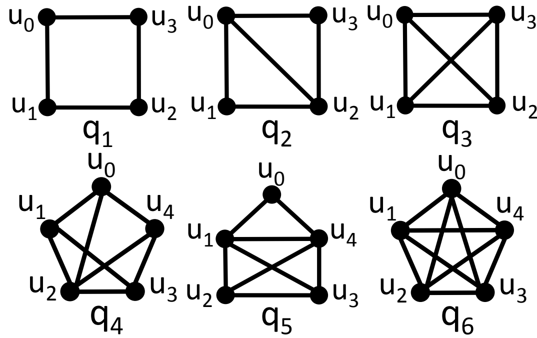


Figure 4: Queries

equal portion of memory for each level $\theta/|V(P)|$ to maintain the partial instances. To generate the partial instances for level $l+1$, we would take a set of partial instances at level l that are estimated to fit into the capacity kept for this level. We repeat this for each level until the instances are found. This can control the size of intermediate result without causing memory overflow.

6 EXPERIMENTS

In this section, we would answer the following questions by conducting extensive experiments.

- Are the proposed techniques effective to improve the performance of PBE?
- Can our proposed solution PBE outperform the existing single-machine approaches?
- How do the parameters, such as the size of data graph, affect the performance of PBE?

6.1 Experimental Setup

Baselines. We mainly focus on comparison with the baselines on CPU and GPU on a single machine. We implement these baselines in our codebase.

- NEMO [26]: The state-of-the-art GPU solution for subgraph enumeration. As the work [26] is originally designed for network motif discovery, we only employ the component for subgraph enumeration in our experimental evaluation.
- GPSM [37]: The state-of-the-art GPU solution for subgraph matching. As subgraph enumeration is defined on unlabeled graphs, we do not execute the filtering phase in GPSM, which generates candidate vertices and edges by labels, but only uses the join phase to enumerate subgraphs.
- CFL [2]: The state-of-the-art index-based CPU solution. As [2] only presents a serial algorithm, we optimize it with multi-threaded support.

Table 2: The datasets used in the experiments.

Dataset	YT	LJ	OR	UK	FR	YH
$\|V\|(\times10^6)$	1.1	3.9	3.0	18.5	65.6	720.2
$\|E\|(\times10^6)$	2.9	34.6	117.1	298.1	1806.0	6434.5
Size (GB)	0.027	0.3	0.9	2.4	14.4	53.3

- VF2 [8, 24]: The widely used nonindex-based CPU solution. We also improve the original implementation with multi-threaded support.

Datasets. We conduct the experiments on the real-world datasets shown in Table 2. The datasets include *youtube* (YT), *livejournal* (LJ), *orkut* (OR), *uk-2002* (UK), *friendster* (FR), *yahoo* (YH).

Queries. We follow most existing works of subgraph enumeration [20, 23, 29, 31] to evaluate the small and dense pattern graphs. Figure 4 lists the pattern graphs tested in the experiments, which are also used in previous studies. In our experiments, we count the number of instances and report the execution time. Each experiment is performed three times, and the average is presented.

Experimental Environment. We conduct the experiments in the following settings.

- For the GPU-based baselines and our approach, we use a machine equipped with 256GB main memory, two NVIDIA TITAN V GPUs (each has 12GB device memory), and two Intel Xeon Gold 6140 CPU processors (each has 18 cores). The programs are compiled with CUDA-10.0 and GCC 7.3.0 with O3 flag.
- For the CPU-based baselines, we use a machine with *four* 10-core Intel Xeon E7-4820 processors (40 cores in total) and 128GB main memory. The programs are compiled with GCC 5.4.0 using O3 flag.

6.2 Effect of Partition Based Framework

In this subsection, we evaluate the effectiveness of our partition based framework. For comparison, we implement two solutions that do not partition the data graph, i.e., UMA and RADS. To access the data graph larger than GPU memory, both solutions rely on the unified memory technology in CUDA. It encapsulates the management of large data and offers a hardware-assisted data access mechanism similar to virtual memory in CPU context, where data is loaded from main memory on demand and cached in GPU memory if possible. To enumerate subgraphs, UMA adopts the same method as the intra-partition search of PBE. It can be considered as a variant of PBE without the inter-partition search by forcing the partition number to be one. In fact, UMA is the direct solution as mentioned in Section 1. RADS is a GPU-based approach adapted from the recent distributed framework [31]. The major process of RADS follows the design of UMA. The

Table 3: Effect of partition based framework. Each entry in the table represents "total time=(communication time, computation time)".

Method	PBE	UMA	RADS
q2	$\mathbf{1.7 \times 10^3 = (1.6 \times 10^3, 5.5 \times 10^1)}$	$1.4 \times 10^4 = (1.4 \times 10^4, 8.0 \times 10^1)$	$7.0 \times 10^3 = (6.8 \times 10^3, 1.6 \times 10^2)$
q3	$\mathbf{1.2 \times 10^3 = (1.2 \times 10^3, 7.5 \times 10^1)}$	$2.0 \times 10^4 = (2.0 \times 10^4, 1.6 \times 10^2)$	$9.7 \times 10^3 = (9.5 \times 10^3, 2.2 \times 10^2)$
q5	$\mathbf{5.8 \times 10^3 = (5.3 \times 10^3, 4.9 \times 10^2)}$	$1.2 \times 10^5 = (1.2 \times 10^5, 1.2 \times 10^3)$	$1.0 \times 10^5 = (9.9 \times 10^4, 1.9 \times 10^3)$
q6	$\mathbf{1.2 \times 10^3 = (1.0 \times 10^3, 2.4 \times 10^2)}$	$2.9 \times 10^4 = (2.9 \times 10^4, 5.3 \times 10^2)$	$1.4 \times 10^4 = (1.3 \times 10^4, 5.0 \times 10^2)$

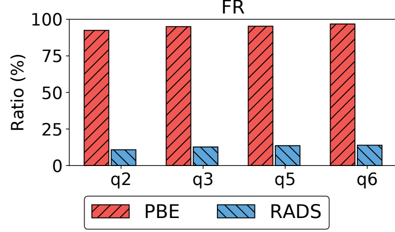


Figure 5: The ratios of the intra-partition instances found by PBE and RADS versus the overall instances.

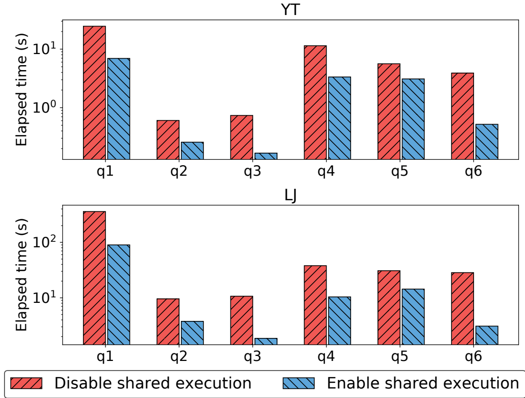


Figure 6: Effect of shared execution.

only difference is that RADS has a local search process to find SOME intra-partition instances before the major process. This process is based on the heuristics that if the border distance (the distance to the border of the graph partition) of a data vertex is longer than the span (the maximum distance to the other pattern vertices) of the first pattern vertex to match, then searching from this data vertex can definitely lead to the intra-partition instances.

As shown in Table 3, in terms of the overall performance, PBE is more than 10 times faster than UMA and RADS for the data graph (FR) that cannot fit into the GPU memory. With the partition based framework, PBE can enumerate intra-partition instances efficiently, as the intra-partition search does not access the data graph in main memory. As would be shown by the profile result below, most instances are intra-partition and thus the overall performance is improved significantly. UMA exhibits the poorest performance, since it can incur data transfer via PCI-e on enumerating *each* subgraph when the data needed is not cached in GPU memory. RADS does not perform well for the same reason as UMA, but it is slightly better than UMA due to the local search process. It can find some intra-partition instances efficiently without causing the PCI-e traffic, which can reduce the workload for the major process and improve the overall performance. However, since the majority of the instances are still enumerated in the major process, the local search process cannot help RADS to achieve comparable performance with PBE.

Besides the overall performance, Table 3 also shows the time breakdown. For each method, we define the communication as the execution that might cause data transfer over PCI-e. Specifically, for PBE we measure the time of Lines 9-10 in Algorithm 3 for inter-partition search; the time at Line 3 of Algorithm 1 is measured for UMA and the major process of RADS. We define the computation time as the total execution time minus the communication time. From the time breakdown, the communication time varies for different methods because of the different strategies to reduce the amount of computation that needs to access the data graph from main memory: the partition based framework for PBE, the hardware-assisted caching in the unified memory for UMA, and the unified memory plus the local search process for RADS. The fact that PBE spends the least communication time confirms that the partition based framework is the most effective strategy.

We profile the intra-partition instances that are found by PBE and RADS and show the results in Figure 5. RADS can only capture a limited number of intra-partition instances, less than 15%. Due to the small-world property of real-world graphs, the border distance is usually small. If using a pattern graph with large diameter, e.g., q2, the heuristics employed by RADS can only identify a few qualified data vertices. In comparison, PBE can capture all intra-partition instances, i.e., over 90% of the instances. As most of the instances are processed in intra-partition search, the workload in inter-partition search is small and the communication cost is significantly reduced.

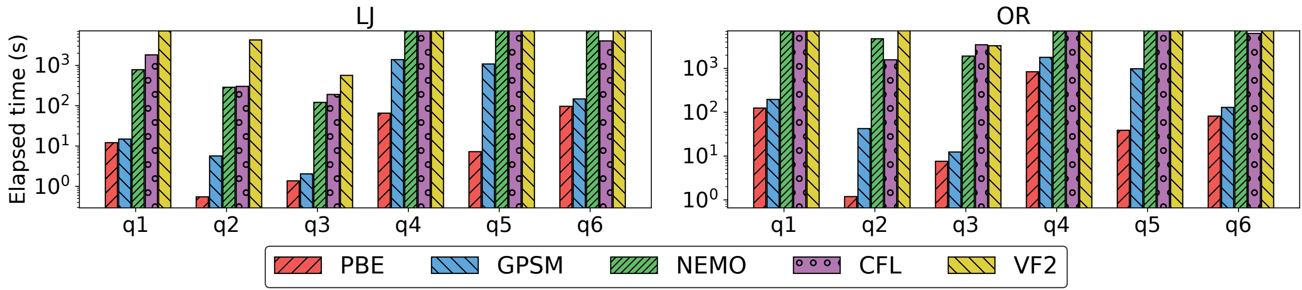


Figure 7: Comparison with baselines by varying queries.

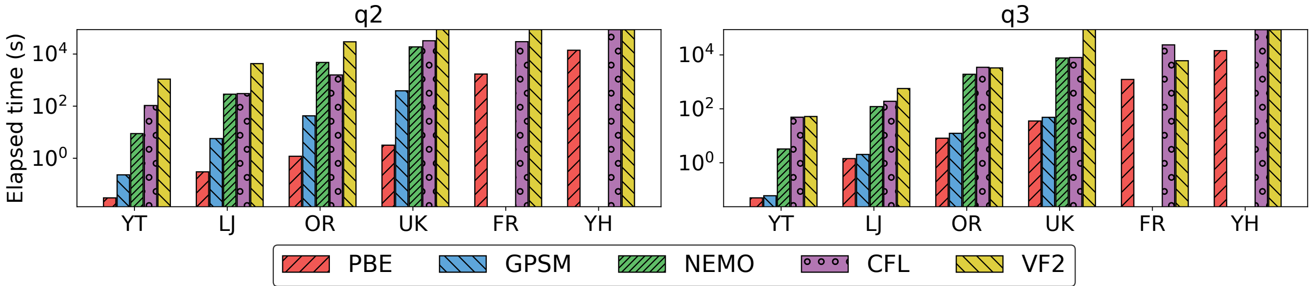


Figure 8: Comparison with baselines by varying datasets. The missing bars of GPSM and NEMO are caused by the inability to process large data graphs.

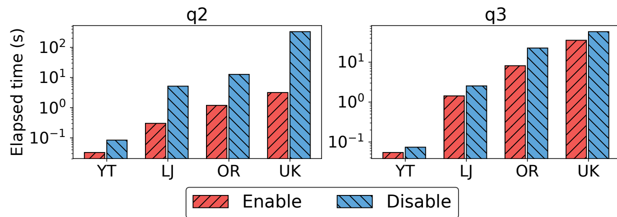


Figure 9: Effect of compression.

6.3 Effect of Shared Execution

In this subsection, we evaluate the effectiveness of shared execution. For comparison, we implement a version of PBE that disables shared execution and searches the inter-partition instances for one search sequence at a time. We preprocess two datasets (YT and LJ) with the number of partitions set to 4. To evaluate the performance of inter-partition search, we only measure the execution time of finding inter-partition instances. Figure 6 shows the performance results of inter-partition search on two implementations of PBE that enable and disable shared execution respectively.

For all queries, shared execution achieves the speedups of 1.4 – 9.2 times. The speedups differ for queries because the amount of sharing varies. For instance, in q6, we can make all inter-partition search sequences in one strict-equivalence group and generate the data sequences for one group altogether. However, in q9, we can only arrange the search sequences into four groups. The number of strict-equivalence groups generated depends on the number of equivalence edge classes in the pattern graph (Lemma 3). Despite different numbers of strict-equivalence groups achieved, shared execution can avoid redundant enumeration costs among multiple search sequences and improve the runtime performance in general cases.

6.4 Comparison with Baselines

In this subsection, we compare the performance of PBE with the single-machine baselines by varying queries (Figure 7) and datasets (Figure 8). Note that we set the running time limits as 2 and 24 hours for the experiments in Figure 7 and Figure 8 respectively.

Varying queries. Compared with the GPU baselines, PBE achieves significantly better performance than NEMO and GPSM. For the datasets that fit into the GPU memory, such as LJ and OR, PBE would directly operate on the data graph and follow the same algorithm as the GPU baselines, i.e., Algorithm 1 to enumerate subgraphs. However, the performances remarkably differ because of the implementation. Based on GPSM, PBE applies a compression technique, which is proposed in [29], to reduce the materialization cost of partial

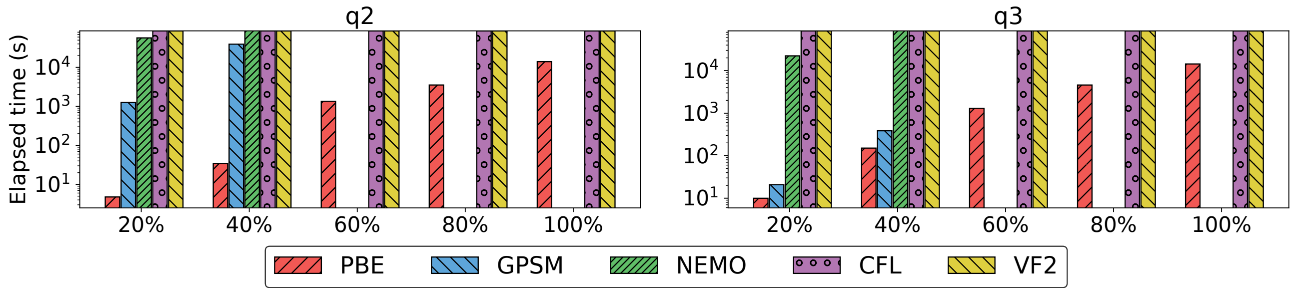


Figure 10: Effect of graph size. The x-axis represents the percentage of data vertices sampled from YH. The missing bars of GPSM and NEMO are caused by the inability to process large data graphs.

instances. It can significantly boost the performance since the number of partial instances is always huge. To demonstrate the effect of compression, we compare PBE with a variant of itself without the compression technique and show the results in Figure 9. As can be seen from the figure, the speedups can be $1.4 - 100$ times. The compression technique is detailed in [29], which is out of scope of this paper. Note that NEMO achieves the worse performance than PBE and GPSM, because it adopts an inefficient method to compute the candidate data vertices (Line 8 of Algorithm 1). To perform the set intersection operation, PBE and GPSM use the binary search to check whether a data vertex exists in an adjacent list, which takes logarithmic time. However, NEMO resorts to the linear scan, which can significantly increase the workload. Such a method might be efficient for small graphs [26], but it leads to poor performance for large graphs.

Compared with the CPU baselines, PBE significantly outperforms CFL and VF2. The speedups can be over three orders of magnitude. Even though we optimize CFL and VF2 with multi-threaded support, they are still slow on the powerful 40-core machine. In comparison, PBE runs on the GPU that has massive (thousands of) thread parallelism and large memory bandwidth. This superior computation power makes the GPU-based approach the better solution for subgraph enumeration.

Varying datasets. On different datasets, PBE consistently outperforms other baselines. Note that for the large graphs including FR and YH, the bars for GPSM and NEMO are missing. This is because they cannot process the data graphs larger than the GPU memory. Such a limitation restricts their application scopes, despite their superior performances to the CPU baselines on the small graphs. The CPU baselines are able to process large graphs, but the restricted computation power can easily lead to poor performance, especially on large graphs. Our GPU solution PBE can overcome the limitations of the GPU and CPU baselines and efficiently scale to large graphs.

Table 4: The statistics of sampled graphs.

% of data vertices	20%	40%	60%	80%	100%
$\lvert V\rvert(\times 10^6)$	80.7	211.6	366.1	538.2	720.2
$\lvert E\rvert(\times 10^9)$	0.3	1.0	2.3	4.1	6.4
Size (GB)	2.5	9.2	19.9	34.7	53.3

6.5 Effect of Graph Size

To evaluate the effect of the size of data graph, we follow the previous works [20, 35] to vary the graph size by sampling subgraphs from a large graph, i.e., YH. According to [20, 35], 20%,40%,60%,80% of the data vertices are randomly sampled and then the induced subgraphs on the selected vertices are generated. We show the statistics of the sampled graphs in Table 4 and present the experimental results in Figure 10. It can be seen that PBE remains efficient and outperforms other baselines as the graph size increases. The bars for GPSM and NEMO are missing when the data vertices sampled are more than 60%, because the sampled graphs are larger than the GPU memory. The CPU baselines always exhibit poor performance regardless of the graph size.

7 CONCLUSION

In this work, we propose a new approach for GPU-accelerated subgraph enumeration that can efficiently scale to large graphs beyond the GPU memory. Our approach divides the data graph into partitions and then search the instances within and across the graph partitions separately. To search the inter-partition instances efficiently, we propose a shared execution approach to reduce the redundant subgraph search among multiple search sequences. The experimental results show that our approach outperforms the existing solutions on the single machine and exhibits competitive performances with state-of-the-art distributed solutions.

Acknowledgements. The project is supported by the grant, MOE2017-T2-1-141, from Singapore Ministry of Education.

REFERENCES

[1] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, 2018.

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

[3] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *ACM SIGMOD Record*, 15(2):61–71, 1986.

[4] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[5] Q. Cai, H. Zhang, W. Guo, G. Chen, B. C. Ooi, K.-L. Tan, and W.-F. Wong. Memepic: Towards a unified in-memory big data management system. *IEEE Transactions on Big Data*, 5(1):4–17, 2018.

[6] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[7] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[8] 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*, 26(10):1367–1372, 2004.

[9] S. Gong, Y. Zhang, and G. Yu. Clustering stream data by exploring the evolution of density mountain. *Proceedings of the VLDB Endowment*, 11(4):393–405, 2017.

[10] S. Gong, Y. Zhang, and G. Yu. Hbp: Hotness balanced partition for prioritized iterative graph computations. In *2020 IEEE 36th International Conference on Data Engineering*. IEEE, 2020.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[12] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *RECOMB*, volume 4453, pages 92–106. Springer, 2007.

[13] W. Guo, Y. Li, M. Sha, and K.-L. Tan. Parallel personalized pagerank on dynamic graphs. *Proceedings of the VLDB Endowment*, 11(1):93–106, 2017.

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

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

[16] D. S. Johnson and M. R. Garey. *Computers and intractability: A guide to the theory of NP-completeness*, volume 1. WH Freeman San Francisco, 1979.

[17] S. R. Kairam, D. J. Wang, and J. Leskovec. The life and death of online groups: Predicting group growth and longevity. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 673–682. ACM, 2012.

[18] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698. ACM, 2017.

[19] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[20] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1231–1245. ACM, 2016.

[21] A. Kyrola, Æ. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 31–46, 2012.

[22] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015.

[23] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

[24] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, volume 6, pages 133–144. VLDB Endowment, 2012.

[25] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 380–389. Springer, 2006.

[26] W. Lin, X. Xiao, X. Xie, and X. L. Li. Network motif discovery: A gpu approach. In *2015 IEEE 31st International Conference on Data Engineering*, pages 831–842, 2015.

[27] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, 2006.

[28] A. Pugliese, M. Bröcheler, V. Subrahmanian, and M. Ovelgönne. Efficient multiview maintenance under insertion in huge social networks. *ACM Transactions on the Web (TWEB)*, 8(2):10, 2014.

[29] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment*, 11(2):176–188, 2017.

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

[31] X. Ren, J. Wang, W.-S. Han, and J. X. Yu. Fast and robust distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 12(11):1344–1356, 2019.

[32] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *Proceedings of the VLDB Endowment*, 2017.

[33] M. Sha, Y. Li, and K.-L. Tan. Gpu-based graph traversal on compressed graphs. In *Proceedings of the 2019 International Conference on Management of Data*, pages 775–792, 2019.

[34] 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*, 1(1):364–375, 2008.

[35] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 625–636. ACM, 2014.

[36] S. Sun and Q. Luo. Efficient parallel subgraph enumeration on a single machine. In *2019 IEEE 35th International Conference on Data Engineering*, 2019.

[37] H.-N. Tran, J.-j. Kim, and B. He. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*, pages 299–315. Springer, 2015.

[38] J. Ugander, L. Backstrom, and J. Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd international conference on World Wide Web*,

pages 1307–1318. ACM, 2013.
[39] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.
[40] J. Wang and J. Cheng. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.
[41] S. Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4), 2006.
[42] Y. Wu, W. Guo, C.-Y. Chan, and K.-L. Tan. Parallel database recovery for multicore main-memory databases. *arXiv preprint arXiv:1604.03226*, 2016.
[43] Y. Wu, W. Guo, C.-Y. Chan, and K.-L. Tan. Fast failure recovery for main-memory dbmss on multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 267–281, 2017.
[44] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 659–670. IEEE, 2017.
[45] S. Zhang, J. He, A. C. Zhou, and B. He. Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *Proceedings of the 2019 International Conference on Management of Data*, pages 705–722, 2019.
[46] 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*, pages 192–203. ACM, 2009.
[47] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.