Efficient Parallel Subgraph Enumeration on a Single Machine

Shixuan Sun Yulin Che Lipeng Wang Qiong Luo
Department of Computer Science and Engineering, Hong Kong University of Science and Technology
{ssunah, yche, lwangay, luo}@cse.ust.hk

***Abstract*—Subgraph enumeration finds all subgraphs in an unlabeled graph that are isomorphic to another unlabeled graph. Existing depth-first search (DFS) based algorithms work on a single machine, but they are slow on large graphs due to the large search space. In contrast, distributed algorithms on clusters adopt a parallel breadth-first search (BFS) and improve the performance at the cost of large amounts of hardware resources, since the BFS approach incurs expensive data transfer and space cost due to the exponential number of intermediate results. In this paper, we develop an efficient parallel subgraph enumeration algorithm for a single machine, named LIGHT. Our algorithm reduces redundant computation in DFS by deferring the materialization of pattern vertices until necessary and converting the candidate set computation into finding a minimum set cover. Moreover, we parallelize our algorithm with both SIMD (Single-Instruction-Multiple-Data) instructions and SMT (Simultaneous Multi-Threading) technologies in modern CPUs. Our experimental results show that LIGHT running on a single machine outperforms existing single-machine DFS algorithms by more than three orders of magnitude, and is up to two orders of magnitude faster than the state-of-the-art distributed algorithms running on 12 machines. Additionally, LIGHT completed all test cases, whereas the existing algorithms fail in some cases due to either running out of time or running out of available hardware resources.**

I. Introduction

Given an unlabeled graph P (called a pattern graph) and an unlabeled graph G (called a data graph), subgraph enumeration finds all subgraphs in G that are isomorphic to P. As one of the fundamental graph analysis operations, it is widely used in real-world applications, such as network motif discovery [26], subgraph frequencies computation [23], the evolution of social networks study [10] and graphlet kernel computation [22].

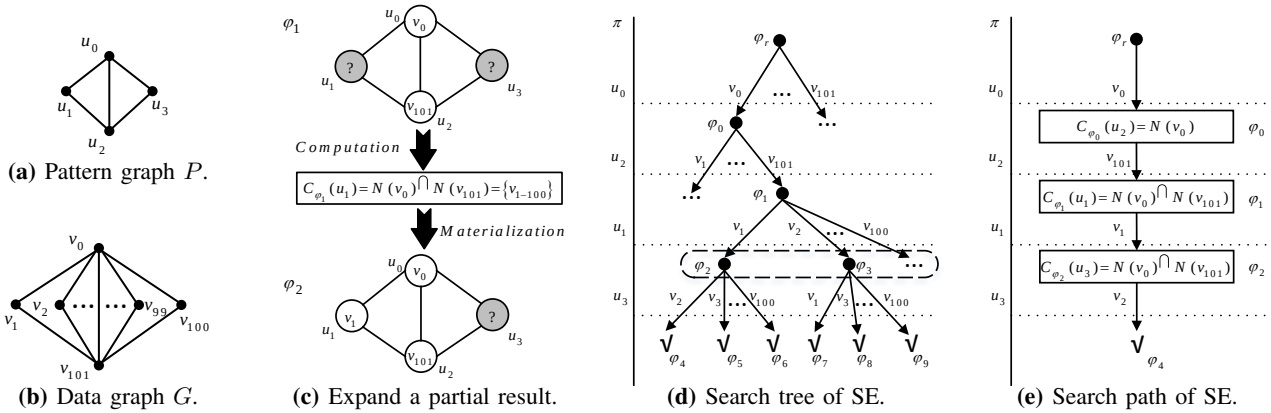
Due to its importance, subgraph enumeration receives a lot of research interests. The algorithms working on a single machine [6], [7], [11] focus on designing filtering rules and optimizing enumeration orders. They all adopt the same in-memory enumeration method that expands partial results vertex-by-vertex along an order of pattern vertices, which is a DFS method. The advantage of the DFS-style approaches is that they consume a small amount of memory during the enumeration. However, due to the lack of the label information, subgraph enumeration is computationally challenging.

Recently, several distributed algorithms [3], [12], [13], [19] have been proposed to handle large data graphs by parallelization. To exploit the parallelism in subgraph enumeration, these algorithms model subgraph enumeration as the *join* problem by decomposing P into small components and joining the results of the components to obtain the final results. Even though this parallel BFS approach with the bulk synchronous parallel (BSP) model has led to significant performance improvement over the algorithms working on a single machine, it has to maintain an exponential number of intermediate results, because the computation of the current iteration depends on the results of the previous one. As a result, the disk I/O and the data shuffling of the intermediate results seriously degrade the performance even with the compression techniques [19]. Additionally, distributed computing resources have higher economic cost as well as maintenance cost than single machines.

Our Approach. Considering the problems in subgraph enumeration, we propose an efficient parallel subgraph enumeration algorithm for a single machine, named **LIGHT**. Our key observation is that there is a large amount of redundant computation (i.e., set intersections) in the enumeration procedure. In the following, we use an example to briefly introduce the basic enumeration method, called **SE**, and illustrate our observation.

Example I.1. *Given P and G in Figures 1a and 1b, the enumeration order π, which is a permutation of pattern vertices, is (u_0, u_2, u_1, u_3). SE recursively expands partial results vertex-by-vertex along π by mapping a pattern vertex to a data vertex at each step to enumerate all results. Figure 1c presents the process of expanding the partial result $\varphi_1 = \{(u_0, v_0), (u_2, v_{101})\}$ by mapping u_1 to a data vertex. The process contains two phases: the computation and the materialization. The computation is to obtain the candidate set of u_1 given φ_1, denoted as $C_{\varphi_1}(u_1)$, by intersecting the neighbor sets of data vertices mapped to the pattern vertices in $N^{\pi}_{+}(u_1)$, which contains the neighbors of u_1 positioned before u_1 in π. In this example, $N^{\pi}_{+}(u_1) = \{u_0, u_2\}$ and $C_{\varphi_1}(u_1) = \{v_{1-100}\}$. After the computation, the materialization is to extend φ_1 by mapping u_1 to data vertices in $C_{\varphi_1}(u_1)$ but not in φ_1. Figure 1c derives φ_2 from φ_1 by mapping u_1 to v_1.*

Figure 1d visualizes the enumeration as a search tree whose nodes and edges denote partial results and mappings respectively, for example, $\varphi_r = \{\}$, $\varphi_0 = \{(u_0, v_0)\}$, $\varphi_1 = \{(u_0, v_0), (u_2, v_{101})\}$ and $e(\varphi_0, \varphi_1)$ denotes (u_2, v_{101}). As the search tree is large, we illustrate part of it for brevity. We focus on φ_1 in the tree. In the materialization phase, SE extends φ_1 by mapping u_1 to candidates that are in $C_{\varphi_1}(u_1)$ but not in φ_1,

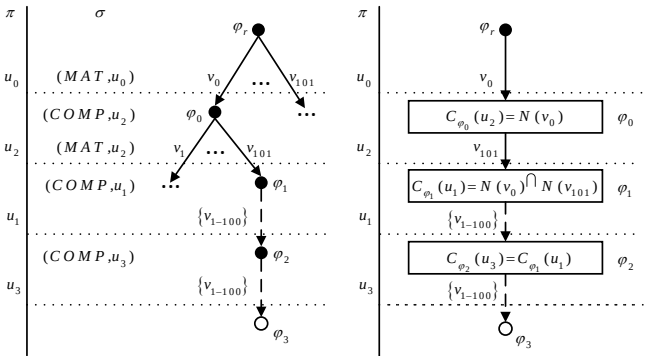


(a) Pattern graph P. **(b)** Data graph G. **(c)** Expand a partial result. **(d)** Search tree of SE. **(e)** Search path of SE.

Fig. 1: Running example of SE.

and generates 100 *new partial results, denoted as* $\Delta(\varphi_1)$*. We find that given the partial results in* $\Delta(\varphi_1)$ *(i.e., the partial results in the dashed rectangle) the same set intersection is repeated in the computation of the candidate sets of* u_3*, for example,* $C_{\varphi_2}(u_3) = C_{\varphi_3}(u_3) = N(v_0) \bigcap N(v_{101})$*, because given any* $\varphi \in \Delta(\varphi_1)$*,* $N_+^{\pi}(u_3) = \{u_0, u_2\}$*,* $\varphi(u_0) = v_0$ *and* $\varphi(u_2) = v_{101}$*. Then, there are* 99 *redundant set intersections among partial results in* $\Delta(\varphi_1)$*. We can eliminate the redundant computation by deferring the materialization of* u_1 *as shown in Figure 2a.*

Next, we focus on a simple search path from the root node to the leaf node, which is shown in Figure 1e. The computation of each candidate set is illustrated in the rectangle. Along this search path, SE conducts 2 *set intersections in total, which can be reduced to only one set intersection. Specifically, SE computes* $C_{\varphi_2}(u_3)$ *by intersecting* $N(v_0)$ *and* $N(v_{101})$*, but the result of intersecting* $N(v_0)$ *and* $N(v_{101})$ *has been obtained and cached in* $C_{\varphi_1}(u_1)$*. If we assign* $C_{\varphi_1}(u_1)$ *to* $C_{\varphi_2}(u_3)$ *as shown in Figure 2b, then we can reduce 1 set intersection in the path.*



(a) Search tree of LIGHT. **(b)** Search path of LIGHT.

Fig. 2: Running example of LIGHT.

In order to solve the problems in Example I.1, LIGHT adopts the *lazy materialization* strategy and the *minimum set cover based candidate set computation* method. Specifically, instead of immediately materializing the pattern vertices after computing their candidate sets, the lazy materialization defers the materialization of pattern vertices until necessary. Furthermore, we convert the candidate set computation into finding the minimum set cover in order to minimize the number of set intersections to compute a candidate set of u by utilizing the candidate sets of vertices before u in the enumeration order.

Finally, we parallelize LIGHT to further improve its performance. We identify two kinds of parallelism in LIGHT: the fine-grained parallelism in set intersections and the coarse-grained parallelism among the partial results. As such, we utilize SIMD (Single-Instruction-Multiple-Data) instructions on vector registers to parallelize set intersections and use SMT (Simultaneous Multi-Threading) techniques in the multi-core processors to process the partial results in parallel.

In summary, we make the following contributions.

- We propose an efficient parallel subgraph enumeration algorithm LIGHT for a single machine.
- We design the lazy materialization technique to reduce redundant computation.
- We propose the minimum set cover based candidate set computation method to further reduce redundant computation.
- We fully exploit both SIMD and SMT in modern CPUs to boost the performance of subgraph enumeration.

We conduct detailed experiments on a variety of real-world datasets to evaluate LIGHT. Experimental results show that LIGHT outperforms DUALSIM [11], the state of the art on a single machine, by up to three orders of magnitude. Compared with the state-of-the-art distributed algorithms SEED [13] and CRYSTAL [19] running on 12 machines, LIGHT achieves a speedup of up to two orders of magnitude. Additionally, LIGHT is the only algorithm completing all test cases.

Paper Organization. Section II presents the preliminaries and the related work. Section III gives an overview of the basic subgraph enumeration algorithm. Sections IV and V introduce the lazy materialization and the minimum set cover based candidate set computation respectively. Section VI presents the method that optimizes the enumeration order. The parallel implementation is discussed in Section VII. We evaluate our algorithm in Section VIII and conclude in Section IX.

II. BACKGROUND

A. Preliminaries

In this subsection, we present the preliminaries and illustrate the frequently used notations in Table I. We focus on the unlabeled undirected graph $g = (V, E)$, where V is a set of vertices and E is a set of edges. Let g_1 and g_2 be two graphs. If $V(g_1) \subseteq V(g_2)$ and $E(g_1) \subseteq E(g_2)$, then g_1 is a *subgraph* of g_2. Given g and $V' \subseteq V(g)$, the *vertex-induced subgraph* of g constructed on V' is denoted as $g[V'] = (V', E')$ where $E' = \{e(u, u') | u, u' \in V' \text{ and } e(u, u') \in E(g)\}$.

Definition II.1. *Subgraph Isomorphism (Match): Given $g = (V, E)$ and $g' = (V', E')$, a subgraph isomorphism from g to g' is an injective function $\varphi : V \to V'$ such that $\forall e(u, u') \in E, e(\varphi(u), \varphi(u')) \in E'$.*

We call a *subgraph isomorphism* from g to g' a *match*. g is isomorphic to g' if and only if there exists a match from g to g', $|V(g)| = |V(g')|$, and $|E(g)| = |E(g')|$. The goal of subgraph enumeration is to find all subgraphs in the data graph G that are isomorphic to the pattern graph P.

An *automorphism* of P is a match from P to itself. Because of the automorphism, a subgraph in G that is isomorphic to P can result in duplicate matches from P to G. In order to eliminate such duplicates, the symmetry breaking technique [7] is proposed, which assigns partial order $<$ to vertices and requires the matches to satisfy that given $u, u' \in V(P)$, if $u < u'$, then $\varphi(u) < \varphi(u')$. Similar to previous work [11]–[13], [19], [21], we rearrange the IDs of vertices in $V(G)$ as *ordered graphs* to preserve the partial order: given any $v, v' \in V(G)$, $v < v'$ if and only if $d(v) < d(v')$ or $d(v) = d(v')$ and $ID(v) < ID(v')$.

If P has only one automorphism, the goal of subgraph enumeration is equivalent to finding all matches from P to G. Otherwise, we adopt the symmetric breaking to avoid duplicates. In this paper, for the ease of analysis, we assume that there is only one automorphism. Thus, the problem addressed in this paper is as follows.

Problem Statement. Given a pattern graph P and a data graph G, find all matches from P to G.

We call the vertices in $V(P)$ and $V(G)$ the pattern vertices and the data vertices respectively. $R(P)$ denotes all the matches from P to G. Thus, our goal is to find $R(P)$. Next, we list the definitions used in this paper.

Definition II.2. *Enumeration Order: Given P, an enumeration order π is a permutation of vertices in $V(P)$. $\pi[i]$ is the ith vertex in π, and $\pi[i : j]$ is the set of vertices from i to j where $1 \leqslant i \leqslant j \leqslant n$.*

Given P and π, $P[\pi[1 : i]]$, which is the vertex-induced subgraph of P constructed on $\pi[1 : i]$, is called the partial pattern graph given π, denoted as P_i^π.

Definition II.3. *Backward Neighbors: Given P and π, the backward neighbors of a pattern vertex u, denoted as $N_+^\pi(u)$, are the neighbors of u positioned before u in π.*

TABLE I: Notations.

Notations	Descriptions
g, P, G	graph, pattern graph and data graph
$V(g)$, $E(g)$	vertex set and edge set of g
n, m, N, M	$\|V(P)\|$, $\|E(P)\|$, $\|V(G)\|$ and $\|E(G)\|$
$d(u)$, $N(u)$	degree and neighbors of u
$e(u, v)$	edge between u and v
π, σ	enumeration order and execution order
$g[V]$	vertex-induced subgraph of g given V
φ	subgraph isomorphism (match)
$C_\varphi(u)$	candidate set of u given φ
$N_+^\pi(u)$	backward neighbors of u in π
P_i^π	vertex-induced subgraph of P on $\pi[1 : i]$
$A^\pi(u)$, $F^\pi(u)$	anchor vertices and free vertices of u given π
$R(P)$	matches from P to G
Φ_u	the set of partial results corresponding to computing candidate sets of u
w_u	the estimated number of set intersections in a computation of the candidate set of u
x	fractional edge cover
ρ, ρ^*	fractional edge cover number and the optimal fractional edge cover number

Definition II.4. *Match Containment: Given P and G, suppose that g and g' are vertex-induced subgraphs of P, and φ and φ' are matches from g and g' to G respectively. φ' contains φ if $\forall (u, v) \in \varphi, (u, v) \in \varphi'$.*

Definition II.5. *Candidate Set: Given P, G, π and $u \in \pi$, suppose that P' is a vertex-induced subgraph of P such that $N_+^\pi(u) \subseteq V(P')$ and φ is a match from P' to G. The candidate set of u given φ, denoted as $C_\varphi(u)$, is $\{v | v \in V(G) \text{ and } \forall u' \in N_+^\pi(u), e(v, \varphi(u')) \in E(G)\}$. $C_\varphi(\pi[1]) = V(G)$.*

Definition II.6. *Min Property: Given a collection S of sets where S has a constant cardinality, if the running time of the set intersection $\bigcap_{s \in S} s$ is proportional to $\min_{s \in S} |s|$, then we say the set intersection has the min property.*

Definition II.7. *Fractional Edge Cover [8]: Given a **hypergraph** $H = (V, E)$ where V is a set of vertices and E is a set of subsets of V, a fractional edge cover of H is a mapping $x : E \to [0, \infty)$ such that $\forall u \in V, \sum_{e \in E, u \in e} x(e) \geqslant 1$.*

Given a *hypergraph* H and a *fractional edge cover* x, the *fractional edge cover number* ρ is the value $\sum_{e \in E(H)} x(e)$. The *optimal fractional edge cover number* ρ^* is the minimum number of all fractional edge covers of H.

Connected Enumeration Order. π is a connected enumeration order if $\forall 1 < i \leqslant n, N_+^\pi(\pi[i]) \neq \emptyset$. In this paper, we assume that P is a connected graph and π is connected. Under this assumption, when we expand φ from P_i^π to G to a new match from P_{i+1}^π to G, the connected order enables us to consider only the neighbors of the data vertices mapped to the backward neighbors of $\pi[i+1]$ instead of scanning all the data vertices, which can reduce the search space.

Graph Storage in Memory. We store the graph with the *compressed sparse row* (CSR) format in memory, which contains an offset array and a neighbors array. Specifically, the neighbors of vertices are **sorted** by their *ID*s, and we consume $O(1)$ time to retrieve the neighbor set of a vertex with the CSR format. Each *ID* is a 32-bit unsigned integer.

Worst-Case Optimal Join (WCOJ). WCOJ algorithms are a class of join algorithms whose running time matches the

maximum output size of a given join query [16]. Recently, researchers derived a tight bound on the output size in terms of the *fractional edge cover*, called the AGM bound [4]. WCOJ algorithms run in time of $O(|IN|^{\rho^*})$, where $|IN|$ is the input size and ρ^* is the optimal fractional edge cover number, from the AGM bound. Example WCOJ algorithms include NPRR [17] and Leapfrog Triejoin [25].

Example II.1. *Given a data graph G, let $M = |E(G)|$. Given P in Figure 1a as the query, a fractional edge cover x of P is $x(e(u_0, u_2)) = 0$ and $x(e(u_0, u_1)) = x(e(u_1, u_2)) = x(e(u_2, u_3)) = x(e(u_0, u_3)) = \frac{1}{2}$. The fractional edge cover number of x is $\sum_{e \in E(P)} x(e) = 2$. Moreover, this value is the optimal fractional edge cover number of P, denoted as ρ^*. Then, $AGM \leqslant M^2$. Therefore, query P produces $O(M^2)$ results. Notably, this bound is tight if you consider G a complete graph on $\sqrt{M}$ vertices. For this complete graph, query P produces $\Omega(M^2)$ results.*

Assumptions. In summary, we have the following assumptions in this paper for the ease of analysis.

1) P is connected and $|V(P)|$ (i.e., n) is a constant value.
2) π is connected (i.e., $\forall 1 \leqslant i \leqslant n$, P_i^π is connected).

B. Related Work

To put our work in context, we categorize the related work into two classes: labeled subgraph enumeration and unlabeled subgraph enumeration.

Labeled Subgraph Enumeration. Because the label information can significantly reduce the search space, most labeled subgraph enumeration algorithms work on a single machine and support larger pattern graphs (tens of vertices) than unlabeled subgraph enumeration. They focus on designing effective filtering strategies (e.g., the *neighborhood label frequency filter*) to prune invalid candidates and optimize the enumeration order to reduce the search space [5], [9].

Unlabeled Subgraph Enumeration. Unlabeled subgraph enumeration can be viewed as a special case of labeled subgraph enumeration that all vertices have the same label. Due to the lack of labels, unlabeled subgraph enumeration has a large search space.

Distributed Algorithms. To handle the large search space, most of the recent work utilizes distributed environments to parallelize the search. Afrati et al. [3] proposed a multiway join based approach to process subgraph enumeration in one map-reduce round. Shao et.al [21] presented an approach based on Giraph. Lai et al. presented TwinTwig [12] and SEED [13] on MapReduce. To reduce the output/shuffle cost, Qiao et al. proposed CRYSTAL [19] to compress the intermediate results.

Algorithms on a Single Machine. N. Chiba et al. [6] proposed an edge-searching based strategy. Some researchers [7], [26] improved the performance by the symmetry breaking technique. Kim et al. [11] designed a disk-based algorithm DUALSIM to handle the data graphs that cannot reside in the memory. The in-memory enumeration procedure of these algorithms all adopts the DFS method. In comparison, Empty-Headed [1], a relational engine for graph processing, utilizes WCOJ algorithms to answer queries.

Other Work. Except the approaches finding the exact solutions, there are also approximation solutions [15] and algorithms working on triangle patterns [18].

In this paper, we focus on unlabeled subgraph enumeration on a single machine that finds exact solutions. More specifically, we improve the performance of the in-memory enumeration procedure, which is widely used in existing unlabeled subgraph enumeration algorithms, by (1) reducing redundant computation and (2) parallelization.

Algorithm 1: SE Algorithm

```
   Input: a pattern graph P and a data graph G
   Output: all matches from P to G
1  begin
2      π ← compute a connected enumeration order of V(P);
3      i ← 1, φ ← {};
4      foreach v ∈ V(G) do
5          Add (π[i], v) to φ;
6          Enumerate(π, φ, i + 1);
7          Remove (π[i], v) from φ;
8  Procedure Enumerate(π, φ, i)
9      if i = |π| + 1 then Output φ, return;
       /* The computation phase.                          */
10     C_φ(π[i]) ← ComputeCandidates(π[i], φ);
       /* The materialization phase.                      */
11     foreach v ∈ C_φ(π[i]) do
12         if v ∉ φ.values then Same as Lines 5-7;
13 Function ComputeCandidates(u, φ)
14     C_φ(u) ← ⋂_{u' ∈ N+_π(u)} N(φ(u'));
15     return C_φ(u);
```

III. Overview & Cost Analysis

In this section, we introduce the basic subgraph enumeration algorithm and analyze it in detail.

A. Basic Subgraph Enumeration Algorithm

The algorithms working on a single machine utilize a method proposed by Ullmann [24] in 1976 to find all matches. Algorithm 1 presents this algorithm called **SE** in this paper. SE first generates a connected enumeration order π, and then expands partial results recursively along π. φ records the mappings from P to G in which $\varphi.keys$ and $\varphi.values$ denote pattern vertices and data vertices in φ respectively. Given a partial result φ whose next pattern vertex in π is u (i.e., $\pi[i]$), the computation phase generates $C_\varphi(u)$ (Line 10) and the materialization phase extends φ by mapping u to candidates that are in $C_\varphi(u)$ but not in $\varphi.values$ (Lines 11-12). In particular, we assume that the set intersection at line 14 satisfies the min property. The enumeration procedure conceptually constructs a search tree on the fly and explores it in the DFS manner. A running example of SE has been introduced in Example I.1.

B. Cost Analysis

Memory. Except P and G, SE maintains a partial result during the enumeration, which consumes $O(n)$ memory. SE keeps a candidate set for each pattern vertex except the first one in the enumeration order. Because there are no duplicate vertices in a candidate set, the candidate sets consume $O(d_{max} \times n)$ space where $d_{max} = \max_{v \in V(G)} d(v)$.

Runtime Guarantees of SE. SE is a specialization of a WCOJ algorithm Leapfrog Triejoin [25]. The enumeration procedure of SE is the same as Leapfrog Triejoin that performs self-joins on $E(G)$, which can be viewed as a relation with the source and the destination vertices as attributes, and requires pattern vertices to map to different data vertices in a result. As such, the running time of SE also matches the AGM bound. In the following, we use an example to illustrate this point.

Example III.1. *Following Example II.1, SE will output $\Omega(M^2)$ results when G is a complete graph on $\sqrt{M}$ vertices. Given φ ($0 \leqslant |\varphi| < 4$), SE derives at most $\sqrt{M}$ new partial results from φ. Therefore, there are at most $\sum_{i=0}^{3} (\sqrt{M})^i$ partial results in the search tree of SE except that at depth 4. For each of these partial results, the computation phase takes $O(\sqrt{M})$ time because of the min property, and the materialization phase also takes $O(\sqrt{M})$ time as a candidate set contains $O(\sqrt{M})$ vertices. The running time of SE is $O(\sqrt{M}\sum_{i=0}^{3} (\sqrt{M})^i) = O(M^2)$. The check at line 12 will not affect the correctness of the analysis, because it can be easily implemented with a hash table and $|\varphi|$ is a constant value.*

Cost Model. Independent of the runtime guarantees, we need a cost model to examine the performance factors and optimize the enumeration order. Given P, G and π, the overall cost T includes the cost of the materialization phase, denoted as T_M, and the cost of the computation phase, denoted as T_C. The DFS-style algorithm expands partial results vertex-by-vertex along π. Then, T_M can be estimated as follows.

$$T_M = \sum_{i=1}^{n} |R(P_i^\pi)|. \tag{1}$$

In order to expand a partial result, the DFS-style algorithm needs to compute the candidate set of the next pattern vertex u in π. Equivalently, the computation of a candidate set of u corresponds to a partial result. Then, we define Φ_u as follows.

Definition III.1. *Given P, G, π and $u \in \pi$, Φ_u contains all the partial results corresponding to computing the candidate set of u. $\Phi_{\pi[1]} = \emptyset$ because $C_\varphi(\pi[1])$ is $V(G)$.*

w_u is the number of set intersections in computing a candidate set of u and α is the average cost of one set intersection. T_C can be calculated as follows.

$$T_C = \alpha \sum_{u \in \pi} w_u |\Phi_u|. \tag{2}$$

Cost of SE. We focus on the cost of the computation phase in SE. $\Phi_u^{(1)}$ and $w_u^{(1)}$ denote the values in SE to differentiate that in LIGHT. We have the following proposition. Due to space limit, we only present the proof sketch.

Proposition III.1. *Given P, G and π, $u = \pi[i+1]$ ($1 \leqslant i < n$). Then, $\Phi_u^{(1)} = R(P_i^\pi)$ in SE.*

Proof. (1) Given $\varphi \in \Phi_u^{(1)}$, φ contains i mappings. Line 14 ensures that $\forall e(u', u'') \in E(P_i^\pi), e(\varphi(u'), \varphi(u'')) \in E(G)$. Line 12 guarantees that there are no duplicate data vertices in φ. Then, φ belongs to $R(P_i^\pi)$. Therefore, $\Phi_u^{(1)} \subseteq R(P_i^\pi)$.

(2) With a constructive proof, we can derive that given $\varphi \in R(P_i^\pi)$, φ can be generated by Algorithm 1. Then, φ corresponds to the computation of the candidate set of u (i.e., $\varphi \in \Phi_u^{(1)}$). Therefore, $R(P_i^\pi) \subseteq \Phi_u^{(1)}$.

With (1) and (2), we get that $\Phi_u^{(1)} = R(P_i^\pi)$ and the proposition is proved. □

Based on Proposition III.1, $|\Phi_u^{(1)}|$ can be calculated as follows. As $|\Phi_{\pi[1]}^{(1)}| = \emptyset$, we set $|\Phi_{\pi[1]}^{(1)}| = 0$.

$$|\Phi_u^{(1)}| = \begin{cases} 0, & u = \pi[1]; \\ |R(P_i^\pi)|, & u = \pi[i+1] \ (1 \leqslant i < n). \end{cases} \tag{3}$$

SE conducts set intersections over the neighbor sets of the data vertices mapped to the backward neighbors of u to compute the candidate set of u. Therefore, $w_u^{(1)}$ can be estimated by Equation 4. We set $w_{\pi[1]}^{(1)} = 0$, since $N_+^\pi(\pi[1]) = \emptyset$.

$$w_u^{(1)} = \begin{cases} 0, & u = \pi[1]; \\ |N_+^\pi(u)| - 1, & u = \pi[i+1] \ (1 \leqslant i < n). \end{cases} \tag{4}$$

IV. Lazy Materialization Subgraph Enumeration

In this section, we propose the lazy materialization subgraph enumeration algorithm LIGHT and give an analysis.

A. General Idea

During the enumeration, SE immediately maps a pattern vertex to its candidates after computing the candidate set. However, the early materialization can incur a large amount of redundant computation. In order to solve the problem, we design the *lazy materialization* that defers the materialization of pattern vertices until necessary. Specifically, given a pattern vertex u, during the enumeration along the given order, we generate the candidate set of u without immediate materialization and do not materialize u until the computation of the candidate sets of some other vertices requiring u.

B. Lazy Materialization

We present the **L**azy mater**I**alization sub**G**rap**H** enumera**T**ion algorithm **LIGHT** in Algorithm 2. After obtaining π, we generate a new order σ for the enumeration instead of π (Lines 2-3). To differentiate from π, we call σ the *execution order*. The ith element in σ is a pair of values, $\sigma_i.mode$ and $\sigma_i.vertex$. $\sigma_i.mode$ specifies the operation, including the computation (COMP in short) or the materialization (MAT in short), in the *Enumerate* procedure, and $\sigma_i.vertex$ is the pattern vertex corresponding to the operation.

Lines 18-29 present the method that generates σ. As the candidate set of $\pi[1]$ is $V(G)$, lines 21-26 loop over all pattern vertices except $\pi[1]$. Given u, we need to ensure that the backward neighbors of u have been materialized before computing the candidate set of u. Therefore, lines 22-26 guarantees that the MAT operations of the backward neighbors of u are positioned before the $COMP$ operation of

u in σ. After lines 27-28, σ contains the MAT operations of all pattern vertices.

LIGHT enumerates all matches from P to G along σ (Lines 9-17). If all pattern vertices have been mapped to data vertices, we output φ (Line 10). If the operation is $COMP$, then we compute $C_\varphi(u)$ and invoke the *Enumerate* recursively if $C_\varphi(u)$ is not empty (Lines 12-14). Otherwise, we materialize u and continue the enumeration (Lines 15-17). The *ComputeCandidates* used in LIGHT will be discussed in Section V. For simplicity, we regard that LIGHT still uses the *ComputeCandidates* in Algorithm 1 until then.

Example IV.1. *Following Example I.1, Figure 2a illustrates the search tree of LIGHT. Given π, LIGHT obtains that $\sigma = ((MAT, u_0), (COMP, u_2), (MAT, u_2), (COMP, u_1), (COMP, u_3), (MAT, u_1), (MAT, u_3))$. Take φ_1 as an example. LIGHT obtains that $C_{\varphi_1}(u_1) = \{v_{1-100}\}$ and skips the materialization of u_1. We mark the edge as dashed line in Figure 2a. At φ_3, u_0 and u_2 have been mapped. Moreover, u_1 and u_3 have obtained their candidate sets $C_{\varphi_3}(u_1) = C_{\varphi_3}(u_3) = \{v_{1-100}\}$. LIGHT materializes u_1 and u_3 to find all matches in the subtree rooted at φ_1, which is equivalent to conduct Cartesian products over $C_{\varphi_3}(u_1)$ and $C_{\varphi_3}(u_3)$. We omit this part for brevity. The number of set intersections in the subtree rooted at φ_1 (exclude φ_1) is reduced from* 100 *in SE to* 1.

C. Analysis

With the same method as in Example III.1, we can show that the running time of LIGHT also matches the AGM bound. Due to page limit, we omit the details. Next, we compare the cost of the computation phase of LIGHT with that of SE under the cost model in Section III-B.

We define the *anchor vertices* and *free vertices* as follows.

Definition IV.1. *Anchor Vertices and Free Vertices: Given P, G, π and σ, anchor vertices of $u \in V(P)$, denoted as $A^\pi(u)$, are the vertices u' satisfy that (1) u' is positioned before u in π; and (2) the MAT operation of u' is before the $COMP$ of u in σ. Free vertices of $u \in V(P)$, denoted as $F^\pi(u)$, are the vertices u' satisfy that (1) u' is positioned before u in π; and (2) the MAT operation of u' is after the $COMP$ of u in σ.*

Based on Definition IV.1, we have the following proposition.

Proposition IV.1. *Given P, G, π and σ, suppose that $u = \pi[i+1]$ $(1 \leqslant i < n)$. $A^\pi(u)$ is a vertex cover of P_i^π and $P[A^\pi(u)]$ is a connected vertex-induced subgraph of P_i^π.*

Moreover, we can get the following proposition similar to Proposition III.1. We omit the proof for brevity.

Proposition IV.2. *Given P, G, π and σ, suppose that $u = \pi[i+1]$ $(1 \leqslant i < n)$ and $\Phi_u^{(2)}$ contains all partial results corresponding to the computation of candidate sets of u in LIGHT. If $\varphi \in \Phi_u^{(2)}$, then $\varphi \in R(P[A^\pi(u)])$ and $\forall u' \in F^\pi(u), C_\varphi(u') \neq \emptyset$; and vice versa.*

Given P and G, both SE and LIGHT adopt the same π and $u = \pi[i+1]$ $(1 \leqslant i < n)$. According to Proposition III.1 and IV.2, $|\Phi_u^{(1)}|$ in SE is equal to $|R(P_i^\pi)|$, and $|\Phi_u^{(2)}|$ in LIGHT is at most $|R(P[A^\pi(u)])|$. Next, we use an example to illustrate this result.

Algorithm 2: LIGHT Algorithm

```
   Input: a pattern graph P and a data graph G
   Output: all matches from P to G
1  begin
2      π ← compute a connected enumeration order of V(P);
3      σ ← GenerateExecutionOrder(π, P);
4      i ← 1, u ← π[i], φ ← {};
5      foreach v ∈ V(G) do
6          Add (u, v) to φ;
7          Enumerate(σ, φ, i + 1);
8          Remove (u, v) from φ;
9  Procedure Enumerate(σ, φ, i)
10     if i = |σ| + 1 then Output φ, return;
11     u ← σ_i.vertex;
12     if σ_i.mode is COMP then
           /* The computation phase.                    */
13         C_φ(u) ← ComputeCandidates(u, φ);
14         if C_φ(u) ≠ ∅ then Enumerate(σ, φ, i + 1);
15     else
           /* The materialization phase.                */
16         foreach v ∈ C_φ(u) do
17             if v ∉ φ.values then Same as Lines 6-8;
18 Function GenerateExecutionOrder(π, P)
19     Set u.visited as false for each u ∈ V(P);
20     σ ← ();
21     foreach u ∈ π along its order in π except π[1] do
22         foreach u' ∈ N_+^π(u) along its order in π do
23             if u'.visited is false then
24                 u'.visited ← true;
25                 Add (MAT, u') to σ;
26         Add (COMP, u) to σ;
27     foreach u ∈ π along its order in π do
28         if u.visited is false then Add (MAT, u) to σ;
29     return σ;
```

Example IV.2. *Follow Example IV.1 and take u_3, which is the fourth vertex in π, as an example. $A^\pi(u_3) = \{u_0, u_2\}$ and $F^\pi(u_3) = \{u_1\}$. $P[A^\pi(u_3)]$ is a connected vertex-induced subgraph of P_3^π. $|\Phi_{u_3}^{(1)}|$ in SE is equal to $|R(P_3^\pi)|$, which is 600. $|\Phi_{u_3}^{(2)}|$ in LIGHT is equal to the number of partial results $\varphi \in R(P[A^\pi(u_3)])$ such that $C_\varphi(u_1) \neq \emptyset$, which is 402. Furthermore, we can extend the partial result $\varphi \in \Phi_{u_3}^{(2)}$ by mapping u_1 to candidates in $C_\varphi(u_1)$ to obtain the partial results in $\Phi_{u_3}^{(1)}$, because $P[A^\pi(u_3)]$ is a vertex-induced subgraph of P_3^π. Note that we do not adopt the symmetry breaking in this example for simplicity.*

In the following, we have a more detailed discussion on the computation cost of SE and LIGHT. Given $\varphi \in \Phi_u^{(2)}$, φ belongs to $R(P[A^\pi(u)])$ and satisfies that $\forall u' \in F^\pi(u), C_\varphi(u') \neq \emptyset$ based on Proposition IV.2. Because $P[A^\pi(u)]$ is a vertex-induced subgraph of P_i^π based on Proposition IV.1, we can extend φ by mapping $u' \in F^\pi(u)$ to candidates in $C_\varphi(u')$ in order to generate the matches from P_i^π to G, denoted as $\Delta(\varphi)$. Let $\Delta(\Phi_u^{(2)})$ denote the matches from P_i^π to G that are generated by extending $\varphi \in \Phi_u^{(2)}$, we have $\Delta(\Phi_u^{(2)}) \subseteq R(P_i^\pi)$. Given $\varphi \in R(P_i^\pi)$, φ contains a match $\varphi' \in R(P[A^\pi(u)])$, as $P[A^\pi(u)]$ is a vertex-induced subgraph of P_i^π. φ' satisfies that $\forall u' \in F^\pi(u), C_{\varphi'}(u') \neq \emptyset$. Thus, $\varphi' \in \Phi_u^{(2)}$ and $\varphi \in \Delta(\Phi_u^{(2)})$, therefore $R(P_i^\pi) \subseteq \Delta(\Phi_u^{(2)})$. Then, $R(P_i^\pi) = \Delta(\Phi_u^{(2)})$.

In case $F^\pi(u) = \emptyset$, we have $R(P_i^\pi) = \Delta(\Phi_u^{(2)}) = \Phi_u^{(2)}$. Let $\mathcal{X}$ contain the pattern vertices u such that $F^\pi(u) \neq \emptyset$.

Given $\varphi \in \Phi_u^{(2)}$, suppose that for each $u' \in F^\pi(u)$, there is by expectation $\Gamma(u')$ candidates in $C_\varphi(u')$ that can be mapped to u' to generate matches in $\Delta(\varphi)$. $\Delta(\Phi_u^{(2)})$ can be estimated as $|\Phi_u^{(2)}| \prod_{u' \in F^\pi(u)} \Gamma(u')$. $T_C^{(1)}$ and $T_C^{(2)}$ denote the cost of the computation phase in SE and LIGHT respectively. With Equation 2 and 3, we derive the following equation to compare $T_C^{(1)}$ and $T_C^{(2)}$.

$$\begin{aligned} T_C^{(1)} - T_C^{(2)} &= \alpha \sum_{u \in \mathcal{X}} w_u^{(1)} (|\Phi_u^{(1)}| - |\Phi_u^{(2)}|) \\ &= \alpha \sum_{u \in \mathcal{X}} w_u^{(1)} |\Phi_u^{(2)}| (\prod_{u' \in F^\pi(u)} \Gamma(u') - 1). \end{aligned} \tag{5}$$

Given a partial result φ whose next pattern vertex in π is u, we extend φ by mapping u to $v \in C_\varphi(u)$. v cannot be mapped if $v \in \varphi.values$. Then, there are at most $|\varphi|$ candidates in $C_\varphi(u)$ that cannot be mapped. Therefore, $\Gamma(u)$ can be estimated as $\max\{0, |C_\varphi(u)| - |\varphi|\} \leqslant \Gamma(u) \leqslant |C_\varphi(u)|$. Because the graphs are unlabeled and P is very small, $|C_\varphi(u)|$ can be large but $|\varphi|$ is small. Therefore, $\Gamma(u)$ is generally greater than 1. The multiplication over $\Gamma(u)$ in Equation 5 indicates that the lazy materialization can reduce a large number of set intersections compared with SE. However, because the value of Γ depends on the properties of input graphs as well as the selected enumeration order, we cannot exactly quantify the differences among $T_C^{(1)}$ and $T_C^{(2)}$. Instead, we conduct experiments to evaluate their performance. The experiment results show that LIGHT can reduce up to 95% set intersections compared with SE (see Section VIII-B1), which confirms our analysis.

We cannot ensure that $\prod_{u' \in F^\pi(u)} \Gamma(u')$ must be greater than 1, because there is no guarantee that given any data graph, $|R(P_i^\pi)|$ is greater than $|R(P[A^\pi(u)])|$, although $P[A^\pi(u)]$ is a connected vertex-induced subgraph of P_i^π.

V. Minimum Set Cover Based Candidate Set Computation

In this section, we introduce the minimum set cover based candidate set computation.

A. General Idea

Inspired by the observation in Example I.1, we can compute the candidate set of a pattern vertex u by utilizing candidate sets of the vertices before u in π. Specifically, we convert this problem into the minimum set cover problem as follows.

- Given P and π, suppose that $u = \pi[i+1]$ $(1 \leqslant i < n)$. The universe U is $N_+^\pi(u)$. A collection S of sets is $\{\{u'\} | u' \in U\} \bigcup \{N_+^\pi(u') | N_+^\pi(u') \subseteq N_+^\pi(u)$ where u' is before u in π $\}$. Identify the smallest sub-collection, denoted as S', of S whose union equals U.

We add $\{\{u'\} | u' \in U\}$ into S to guarantee that there must be sub-collections of S whose union can cover U. When adding $N_+^\pi(u')$ into S, we require that $N_+^\pi(u') \subseteq N_+^\pi(u)$ and u' is before u in π. This requirement ensures that $C_\varphi(u) \subseteq C_\varphi(u')$ and $C_\varphi(u')$ has been obtained when computing $C_\varphi(u)$.

Algorithm 3: Minimum Set Cover based Candidate Set Computation

```
1  Procedure GenerateOperands(π, P)
2      Set u.K1 and u.K2 as empty for each u ∈ V(P);
3      for i ← 2 to n do
4          u ← π[i], U ← N+π(u), S ← {{u'} | u' ∈ U};
5          for j ← 1 to i − 1 do
6              u' ← π[j];
7              if N+π(u') ⊆ N+π(u) then S ← S ∪ {N+π(u')};
8          Identify the minimum set cover S' of S that covers U;
9          foreach s ∈ S' do
10             if |s| = 1 then Add u' in s into u.K1;
11             else Add u' before u that N+π(u') = s into u.K2;
12 Function ComputeCandidates(u, φ)
13     Cφ(u) = (⋂_{u'∈u.K1} N(φ(u'))) ⋂ (⋂_{u'∈u.K2} Cφ(u'));
14     return Cφ(u);
```

After identifying S', we process the elements s in S' as follows: If $|s| = 1$, then add the pattern vertex in s into a set K_1; otherwise, add u' before u such that $N_+^\pi(u') = s$ into K_2. If multiple pattern vertices satisfy this condition, we select one randomly.

If $|s| = 1$, then the pattern vertex in s must be an anchor vertex of u, because the construction method of S ensures that the vertices in the sets of S must be the backward neighbors of u or that of the pattern vertices before u in π. Therefore, the vertices in K_1 have been mapped to data vertices when computing the candidate set of u during the enumeration. Furthermore, the candidate sets of the vertices in K_2 have been generated, because they are positioned before u in π. Then, we can compute $C_\varphi(u)$ by Equation 6.

$$C_\varphi(u) = (\bigcap_{u' \in K_1} N(\varphi(u'))) \bigcap (\bigcap_{u' \in K_2} C_\varphi(u')). \tag{6}$$

B. Candidate Set Computation

Algorithm 3 presents the minimum set cover based candidate set computation method. Given a pattern vertex u, we call the vertices in $u.K_1$ and $u.K_2$ the operands of u. *GenerateOperands* takes π and P as input and generates the operands of pattern vertices except $\pi[1]$, because the candidate set of $\pi[1]$ is $V(G)$. Lines 12-14 presents the method that computes $C_\varphi(u)$ based on the operands of u. Because the operands of pattern vertices are determined by the enumeration order, LIGHT invokes *GenerateOperands* to obtain the operands of pattern vertices before the enumeration process. During the enumeration process, LIGHT directly invokes the *ComputeCandidates* function to compute the candidate sets.

Example V.1. *Following Example IV.1, let us consider u_3. $U = N_+^\pi(u_3) = \{u_0, u_2\}$ and $S = \{\{u_0\}, \{u_2\}, \{u_0, u_2\}\}$. We obtain that S' is $\{\{u_0, u_2\}\}$. Therefore, $u.K_1 = \emptyset$ and $u.K_2 = \{u_1\}$, as $N_+^\pi(u_1) = \{u_0, u_2\}$. During the enumeration, LIGHT assigns $C_{\varphi_1}(u_1)$ to $C_{\varphi_2}(u_3)$ as shown in Figure 2b, which reduces 1 set intersection in the search path.*

C. Analysis

Given $u = \pi[i]$ in the loop of lines 3-11 in Algorithm 3, line 8 dominates the cost, because the minimum set cover problem is NP-hard. There are at most $2(i-1)$ elements in S. Then, line 8 consumes $O(2^{2(i-1)})$ time. Therefore, the time complexity

of *GenerateOperands* is $O(\sum_{i=2}^{n} 2^{2(i-1)}) = O(4^n)$. As P is small, *GenerateOperands* can process it very efficiently.

The number of set intersections in one computation of the candidate set of u with Algorithm 3, denoted as $w_u^{(2)}$, can be calculated as follows.

$$w_u^{(2)} = \begin{cases} 0, & u = \pi[1]; \\ |u.K_1| + |u.K_2| - 1, & u = \pi[i+1] \ (1 \leqslant i < n). \end{cases} \tag{7}$$

Because the operands of pattern vertices are obtained based on the minimum set cover, we have Proposition V.1.

Proposition V.1. *Given P and π, $\forall u \in \pi$, $w_u^{(2)} \leqslant w_u^{(1)}$.*

VI. Optimizing Enumeration Order

The enumeration order π has an important impact on the performance of LIGHT. Given P, G, π and σ, the overall cost of LIGHT includes the cost of the materialization and the cost of the computation. The order of materializing pattern vertices in LIGHT follows the sequence of their MAT operations in σ, which is denoted as π'. Then, the cost of the materialization can be estimated as $\sum_{i=1}^{n} |R(P_i^{\pi'})|$. According to Proposition IV.2, $|\Phi_u^{(2)}|$ is at worst equal to $|R(P[A^\pi(u)])|$. Therefore, we estimate $|\Phi_u^{(2)}|$ as $|R(P[A^\pi(u)])|$. Then, the overall cost can be estimated by Equation 8.

$$T = \alpha \sum_{u \in \pi} w_u^{(2)} |R(P[A^\pi(u)])| + \sum_{i=1}^{n} |R(P_i^{\pi'})|. \tag{8}$$

In order to compute Equation 8, we need to estimate the value of its parameters. The value of $w_u^{(2)}$ can be computed by Equation 7. Next, a challenging problem is to estimate $|R(P')|$ where P' is a subgraph of P. This problem is also very important in generating an efficient join plan in distributed algorithms [3], [12], [13], [19], [21]. Instead of inventing a new method, we utilize the estimation approach proposed by SEED [13], which is shown to be effective in real-world data graphs. Specifically, this method calculates an expand factor for each edge of P' by simulating the construction of the partial results in $R(P')$ through extending one edge at each step. Finally, we estimate the value of α. The cost of set intersections depends on the sizes of input sets. Because pattern vertices can be mapped to different data vertices, we address this problem by considering the relationship between α and the expand factors. As the expand factor of an edge is at most the expected degree of the data vertices mapped to the incident vertices of the edge, we simply estimate the value of α as the maximum value of all expand factors. We take the maximum value to give a higher weight to the cost of the computation than that of the materialization, because the cost of a set intersection is much higher than mapping a pattern vertex to a data vertex in practice.

As P is very small, LIGHT simply enumerates all the connected orders of $V(P)$ and selects the order with the minimum value of Equation 8 as the enumeration order. Furthermore, we use partial orders to break ties by prioritizing the order that puts the vertices constrained by partial orders before other vertices. Additionally, we reduce the number of orders by the symmetry breaking: given $u_i, u_j \in V(P)$, if $u_i < u_j$, then u_i must be positioned before u_j in π.

VII. Parallel Implementation

Because of the large search space in subgraph enumeration, we parallelize LIGHT with SIMD and SMT in modern CPUs to improve its performance.

A. SIMD Parallelization

The min property of the set intersection can be achieved by storing sets as hash tables and scanning the smallest set to check the other sets. However, hash tables incur random access patterns in practice, which is costly in memory hierarchies [25]. Another alternative is to implement pair-wise set intersections with SIMD [1], [14]. In this paper, we adopt a hybrid set intersection algorithm implemented with SIMD, which is shown to be effective in graph processing [1].

Algorithm 4: Set Intersection Algorithm

```
1 Function Hybrid(S1, S2)
2     if |S1|/|S2| < δ and |S2|/|S1| < δ then return Merge(S1, S2);
3     else return Galloping(S1, S2);
```

Algorithm 4 presents the *Hybrid* algorithm that takes two **sorted** sets as input and outputs their intersection. If the sizes of S_1 and S_2 are similar, *Hybrid* invokes the *Merge* function that loops over S_1 and S_2 to find their common elements, whose time complexity is $O(|S_1| + |S_2|)$. *Merge* performs poorly when the sizes of the two sets differ greatly, which we call the *cardinality skew*. We use the *Galloping* algorithm [1] to handle this skew. Suppose that $|S_1| < |S_2|$. The time complexity of *Galloping* is $O(|S_1| \times log(|S_2|))$. δ is the threshold for the size ratio, which is configured as 50 in our experiments based on a previous performance study [14].

B. SMT Parallelization

To utilize SMT, we implement LIGHT with the parallel DFS paradigm [20]. Because partial results can be expanded independently to find the matches containing them, we take the partial results as the parallel tasks and each worker (i.e., thread) expands the assigned partial results in DFS independently. In a shared-memory environment, we keep the load balance by the *sender-initiated work stealing* strategy with a global concurrent queue for communication. Specifically, workers monitor the global concurrent queue and the status of the other workers. If a busy worker finds that there are idle workers and the queue is empty, then it sends part of its tasks to the queue and wakes up the idle workers to fetch the tasks. With this approach, idle workers are able to almost immediately acquire tasks by the donation of busy workers [2]. Compared with the parallel BFS approach, our parallel DFS implementation does not maintain an exponential number of intermediate results, so our algorithm can execute on a single machine with the limited memory space. Suppose that there are k workers. Each worker maintains a partial

result and keeps a candidate set for each pattern vertex. LIGHT consumes at most $O(k \times n \times d_{max})$ memory where $d_{max} = \max_{v \in V(G)} d(v)$.

VIII. EXPERIMENTS

In this section, we evaluate the effectiveness of the techniques in LIGHT and compare it with existing algorithms.

A. Experimental Setup

Algorithms Under Study. We compare the following unlabeled subgraph enumeration algorithms in our experiments.

- LIGHT: the algorithm proposed in this paper.
- DUALSIM [11]: the state-of-the-art parallel algorithm working on a single machine.
- SEED [13]: the state-of-the-art distributed algorithm working in MapReduce with *clique-star* as join units.
- CRYSTAL [19]: the state-of-the-art distributed algorithm working in MapReduce with *core-crystal* as join units.

We acknowledge that comparing a shared-memory parallel algorithm with distributed algorithms is not exactly apple-to-apple, and that MapReduce-based solutions have goals such as scalability, ease of programming, and reliability than merely latency. Nevertheless, we compare with them with a focus on the space cost of the BFS approach adopted by both SEED and CRYSTAL.

Furthermore, we evaluate the following algorithms to examine the effectiveness of the individual techniques in LIGHT. The first group is to evaluate the strategies that reduce redundant computation. In particular, we compare with EmptyHeaded [1] and CFL [5] to study whether and how the join plan optimizer in the relational engine and the pruning strategies as well as the enumeration order optimization in labeled subgraph enumeration can benefit unlabeled subgraph enumeration. In order to exclude the effect of the parallelization, these algorithms including ours execute in serial without SIMD. The algorithms under comparison are as follows.

- EH: EmptyHeaded [1], a relational engine for graph processing that answers queries with WCOJ algorithms.
- CFL [5]: the state-of-the-art labeled subgraph enumeration algorithm.
- SE: Algorithm 1, which is the baseline algorithm.
- LM: LIGHT with the **L**azy **M**aterialization strategy only.
- MSC: LIGHT with the **M**inimum **S**et **C**over based candidate set computation method only.
- LIGHT: LIGHT with both the lazy materialization and the minimum set cover based candidate set computation.

The second group is to examine the effectiveness of parallelization. We evaluate the SIMD parallelization by varying the set intersection method in LIGHT. We use AVX2, a SIMD instruction set that can manipulate 256-bit data in one instruction, to implement the set intersection methods. These algorithms execute in one thread. Next, we vary the number of threads to evaluate the SMT parallelization. The algorithms under comparison are as follows.

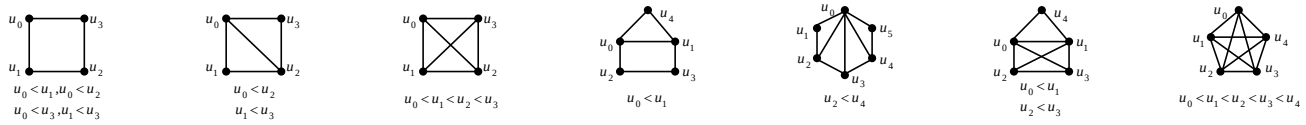
- Merge: LIGHT with the *Merge* set intersection.
- MergeAVX2: LIGHT with the *Merge* set intersection implemented with AVX2.
- Hybrid: LIGHT with the *Hybrid* set intersection.
- HybridAVX2: LIGHT with the *Hybrid* set intersection implemented with AVX2.

Additionally, we test a naive distributed version of LIGHT by storing the entire data graph on each machine and dividing the search space by partitioning $C_\varphi(\pi[1])$ (i.e., $V(G)$) evenly on each machine. However, the speedup is very limited because of the load imbalance among machines, as our naive distributed solution is missing (1) the estimation of workload given a partition of the candidate set and (2) an efficient dynamic load balancing strategy in the shared-nothing environment. We also set up Hadoop in the pseudo distributed mode on a single machine and execute both SEED and CRYSTAL on this machine. Both SEED and CRYSTAL run much slower than LIGHT under this configuration, because (1) Both SEED and CRYSTAL have intensive disk I/O operations to read/write intermediate results; and (2) The novel techniques in SEED and CRYSTAL, such as the graph partitioning and the bloom filtering, for the shared-nothing environment are unnecessary and cause extra overhead on the single machine. We omit the results for brevity.

Experimental Environment. We implement all our algorithms in C++. We obtain the binary of DUALSIM and the source code of CFL, which are both implemented in C++. We extract the generated C++ code of the queries from EmptyHeaded, and modify it to support the symmetry breaking and require pattern vertices to map to distinct data vertices in a result. Except DUALSIM, which is binary executable, we compile the source code of all other algorithms with icpc 16.0.0. We perform the experiments on these algorithms on a machine equipped with 20 cores (2 Intel Xeon E5-2650 v3 @ 2.30GHz CPUs), 64GB RAM and 1TB HDD. We configure the memory buffer in DUALSIM as 32GB so that DUALSIM conducts the enumeration in memory.

We set up Apache Hadoop 2.7.4 on a cluster with 12 machines including 1 master and 11 slaves. Each node is equipped with 16 cores (2 Intel Xeon E5-2630 v3 @ 2.40GHz CPUs), 64GB RAM and 1TB HDD. The HDFS has around 6TB available space in total with the replication factor as two. Each slave has at least 512GB free space. We allocate a JVM heap space of 3584MB for each mapper and 5120MB for each reducer, and we allow at most ten mappers/reducers to run concurrently in each machine. The I/O sort size is 512MB. We obtain the JAVA code of SEED and CRYSTAL from their authors and execute them on this cluster.

Data Graphs. We select 6 real-world datasets that are widely used in previous work [11]–[13], [19], [21]. *lj*, *ot* and *fs* are downloaded from SNAP (http://snap.stanford.edu), *yt* is downloaded from KONECT (http://konect.uni-koblenz.de), and *eu* and *uk* are downloaded from WEB (http://law.di.unimi.it). As shown in Table II, N scales from less than one million to tens of millions and M scales from millions to billions. The graphs stored in the CSR format consume a small amount of memory space.



(a) P_1. **(b)** P_2. **(c)** P_3. **(d)** P_4. **(e)** P_5. **(f)** P_6. **(g)** P_7.

Fig. 3: Pattern graphs.

TABLE II: Properties of real-world datasets.

Dataset	Name	N (million)	M (million)	Memory (GB)
youtube	*yt*	3.22	9.38	0.09
eu-2005	*eu*	0.86	19.24	0.15
live-journal	*lj*	4.85	68.48	0.53
com-orkut	*ot*	3.07	117.19	0.89
uk-2002	*uk*	18.52	298.11	2.30
friendster	*fs*	65.61	1,806.07	13.71

Pattern Graphs. Figure 3 shows the pattern graphs used in our experiments which are the same as those in SEED [13]. n varies from 4 to 6, and m varies from 4 to 10. Partial orders for the symmetry breaking are listed under the pattern graphs.

Metrics. The *execution time* reported in our experiments is the elapsed time on the enumeration. The time limit for processing a pattern graph is 24 hours. In our bar charts of the execution time, bars labeled INF are the algorithms that run out of time limit (OOT), and missing bars are the algorithms that run out of the disk/memory space (OOS). Similar to previous work [11]–[13], [21], all algorithms under study enumerate the matches without storing them into the file system.

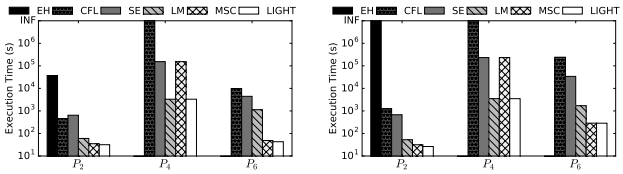
B. Evaluation of Individual Techniques

To obtain sufficient results for comparison, we extend the time limit for each query to 72 hours in this subsection. For brevity, we present the experiment results of P_2, P_4 and P_6 on *yt* and *lj*.

1) Reducing Redundant Computation: Before the comparison, we first briefly introduce the general idea of EH and CFL. EH is a generic graph processing engine that can answer a variety of graph queries, such as pattern queries, PageRank and Single-Source Shortest Paths. In particular, given P and G, EH first divides P into small components (i.e., subgraphs), then finds the matches of each component with a similar logic to SE, and finally joins the matches of the components. CFL partitions the search procedure into two phases: the preprocessing and the enumeration. The preprocessing builds a light-weight index, which contains the data vertices that can be mapped to the pattern vertices, and then generates an enumeration order based on the index. The enumeration procedure recursively finds all results based on the index, which is also similar to SE. In the experiments, the enumeration orders of SE, LM, MSC and LIGHT are the same, which are denoted as π^1. We denote the enumeration order of CFL and EH as π^2 and π^3 respectively. Specifically, $\pi^1(P_2) = \pi^2(P_2) = (u_0, u_2, u_1, u_3)$, $\pi^1(P_4) = (u_0, u_1, u_4, u_2, u_3)$, $\pi^2(P_4) = (u_0, u_2, u_4, u_1, u_3)$, and $\pi^1(P_6) = \pi^2(P_6) = (u_0, u_1, u_2, u_3, u_4)$. EH generates $\pi^3(P_2) = (u_1, u_3, u_0, u_2)$. For P_4, EH divides it into two vertex-induced subgraphs P_4' and P_4'' where $V(P_4') = \{u_0, u_1, u_3, u_4\}$ and $V(P_4'') = \{u_0, u_2, u_3\}$, and finds $R(P_4')$ and $R(P_4'')$ with $\pi^3(P_4') = (u_0, u_3, u_4, u_1)$ and $\pi^3(P_4'') = (u_0, u_3, u_2)$. For P_6, EH also obtains two vertex-induced subgraphs P_6' and P_6'' where $V(P_6') = \{u_0, u_1, u_2, u_3\}$ and $V(P_6'') = \{u_0, u_1, u_4\}$, and finds $R(P_6')$ and $R(P_6'')$ with $\pi^3(P_6') = (u_0, u_1, u_2, u_3)$ and $\pi^3(P_6'') = (u_0, u_1, u_4)$ respectively. Figures 4 and 5 illustrate the experiment results of the execution time and the number of set intersections of these algorithms. If a query cannot be completed due to OOT or OOS, then there is no experiment result of the number of set intersections of this query.

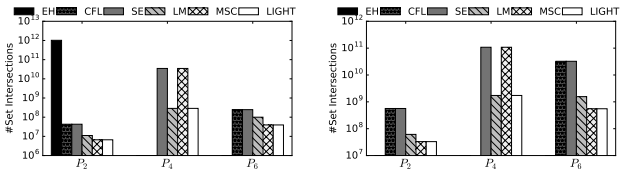
We first compare SE with EH and CFL. EH spends more time on P_2 on *yt* than SE, because the enumeration order of EH is not connected, which results in much more number of set intersections than SE as shown in Figure 5a. The number of set intersections of EH is around 10^4 times more than that of SE. The execution time of EH is only 60 times longer than that of SE, because *yt* is very sparse, in which over 83% vertices have degrees less than 10. Then, a large number of set intersections is among small sets, which accounts for a small portion of the execution time. EH fails on P_2 on *lj* due to OOT. In order to find $R(P_4)$, EH has to store $R(P_4')$ and $R(P_4'')$ in memory before joining them. As a result, EH fails on P_4 on both *yt* and *lj* due to running out of memory. For the same reason, EH fails on P_6. CFL and SE generate the same enumeration order on both P_2 and P_6 and conduct the same number of set intersections. However, CFL spends less time than SE on P_2 on *yt*, but more time on the other three cases, because CFL and SE use different set intersection methods. Specifically, CFL computes set intersections by looping over the smaller set to check whether its elements exist in the other one, which is efficient to cope with the cardinality skew. In contrast, the *Merge* method in SE works well when the sets have a similar size. P_2 on *yt* has much more portion of the number of set intersections on the sets that are cardinality skew than other three cases. CFL fails on P_4 on both *yt* and *lj* due to its enumeration order. The experiment results suggest that SE generates more effective enumeration orders than EH and CFL, and the filtering methods (e.g., the light-weight index) designed for labeled subgraph enumeration are often ineffective on unlabeled graphs.

Next, we compare SE with the variants of LIGHT. By the lazy materialization, LM reduces a large number of set intersections compared with SE and runs much faster than SE. MSC improves the performance of SE on P_2 and P_6 as well, because the number of set intersections along a search path is reduced from 2 to 1 on P_2, and 4 to 2 on P_6. As a result, the total number of set intersections is significantly reduced. Furthermore, the number of set intersections of MSC is less than that of LM on P_2 and P_6, because MSC reduces the number of set intersections in one computation of the candidate set of u_3 in P_2 and u_4 in P_6 from 1 to 0. However, for P_4, MSC cannot reduce the number of set intersections.



(a) *yt* **(b)** *lj*

Fig. 4: Execution time comparison.

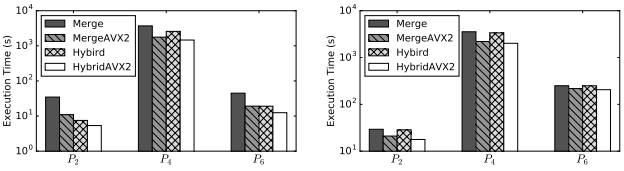


(a) *yt* **(b)** *lj*

Fig. 5: Comparison of number of set interesections.

Consequently, the performance of MSC is similar to SE on P_4, but much slower than LM. The experiment results demonstrate the effectiveness of the lazy materialization and the minimum set cover based candidate set computation respectively. Furthermore, the two techniques are orthogonal, because the lazy materialization aims to reduce the number of times of computing the candidate sets, whereas the minimum set cover based candidate set computation tries to reduce the number of set intersections in a computation of the candidate set.

2) Parallelization: As shown in Figure 6, Hybrid is faster than Merge in all cases. Hybrid runs 1.43-4.62X faster than Merge on *yt*, whereas it only achieves a speedup of 1.01-1.06X on *lj*, because the number of the *Galloping* search accounts for a small portion of the number of set intersections on *lj* as shown in Table III. With SIMD, HybridAVX2 and MergeAVX2 gain a speedup of 1.2-1.8X and 1.2-3.2X. Overall, HybridAVX2 runs 1.2-6.5X times faster than Merge in the six test cases.



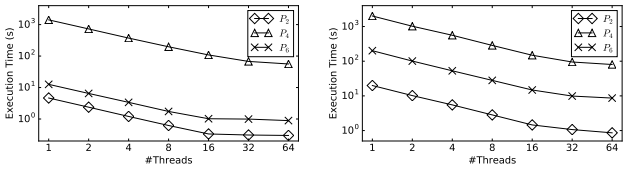
(a) *yt* **(b)** *lj*

Fig. 6: Execution time with different set intersection methods.

TABLE III: Percentage of the *Galloping* search.

Dataset	*yt*			*lj*		
Pattern	P_2	P_4	P_6	P_2	P_4	P_6
Percentage	34.8%	35.9%	8.1%	1.1%	2.1%	0.7%

Figure 7 presents the execution time of LIGHT, which adopts the *Hybird* set intersection implemented by AVX2, with the number of threads varied from 1 to 64. When the number of threads varies from 1 to 16, LIGHT achieves almost linear speedup, but the speedup becomes limited when scaling the number of threads up to 32 and 64. This difference is because there are only 20 physical cores in the machine. Generally, LIGHT obtains more speedup when the execution time is long, because the benefit of the multi-threading is small compared with its overhead when the execution time is very short. For example, LIGHT with 64 threads achieves a speedup of around 15.5X on P_2 on *yt*, and the speedup is up to 25X on P_4 on *yt* (A speedup of more than 20X on 20 physical cores is achieved because of the *hyper threading* in modern CPUs). Therefore, in our experiments, we execute LIGHT with 64 threads to fully utilize the multi-core parallelism.



(a) *yt* **(b)** *lj*

Fig. 7: Execution time with the number of threads varied.

3) Summary: Table IV summarizes the performance improvement with the techniques in this paper. T_{SE} is the execution time of SE. T_{LIGHT} is the execution time of LIGHT without any parallelization, while $T_{LIGHT+P}$ is that with parallelization (*Hybrid* implemented with AVX2 and 64 threads). We parallelize SE with the same method as LIGHT, and this parallelization greatly improves the performance of SE. Nevertheless, on complex query patterns, SE+P still takes a long time. By adopting the strategies that reduce redundant computation, LIGHT without any parallelization significantly outperforms SE as well as SE+P on P_4 and P_6. In total, LIGHT+P is over three orders of magnitude faster than SE ($T_{SE}/T_{LIGHT+P}$).

TABLE IV: Comparison with SE (seconds).

Dataset	*yt*			*lj*		
Pattern	P_2	P_4	P_6	P_2	P_4	P_6
T_{SE}	645	176,181	4,448	677	232,800	34,090
T_{SE+P}	22	4,034	115	15.9	6,949	1,425
T_{LIGHT}	31	3,309	43	26	3,497	285
$T_{LIGHT+P}$	0.3	56	0.9	0.9	80	8.7
Speedup	2,150X	3,146X	4,942X	752X	2,910X	3,918X

4) Memory Consumption: For brevity, Table V only lists the memory consumption of the candidate sets on P_5 with 64 threads, because P_5 has more vertices than the other pattern graphs and our experiments execute LIGHT with 64 threads at most. The memory cost of data graphs has been presented in Table II. As shown in Table V, the candidate sets consume very small memory space, which shows that the parallel DFS method has a good space complexity.

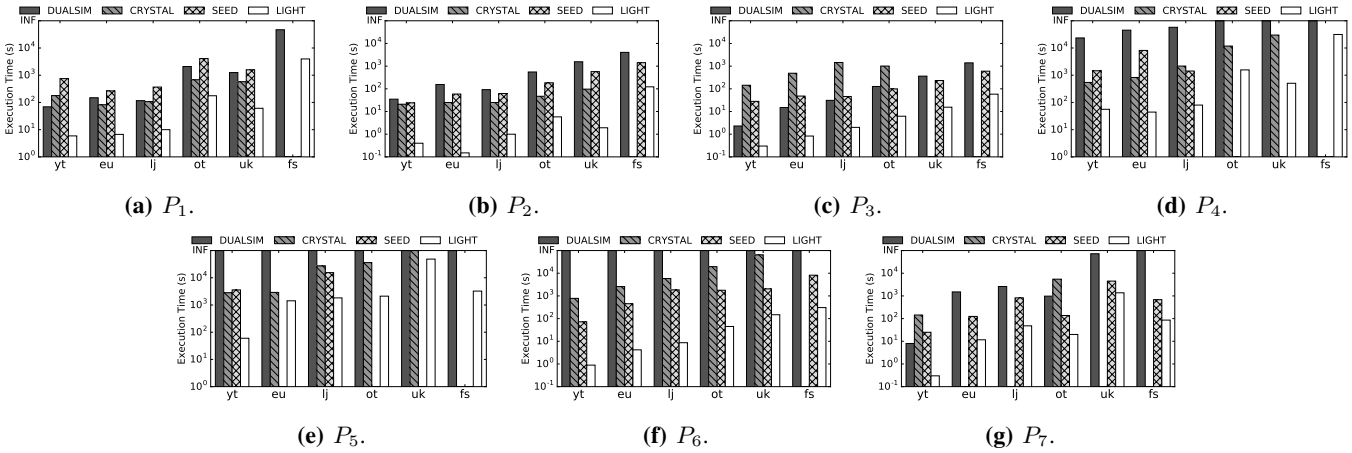
TABLE V: Memory consumption of candidate sets on P_5.

Dataset	*yt*	*eu*	*lj*	*ot*	*uk*	*fs*
Memory (GB)	0.123	0.090	0.022	0.048	0.239	0.008

C. Overall Performance Comparison

We compare LIGHT with the state of the art - DUALSIM on a single machine and both SEED and CRYSTAL on a cluster of 12 nodes. We keep in mind the performance overhead of the MapReduce-based distributed algorithms, and focus on their space cost.

As SEED and CRYSTAL preprocess data graphs quickly, we do not report the preprocessing time. An exception is



(a) P_1. **(b)** P_2. **(c)** P_3. **(d)** P_4.

(e) P_5. **(f)** P_6. **(g)** P_7.

Fig. 8: The execution time of LIGHT, DUALSIM, SEED and CRYSTAL on the real-world datasets.

that CRYSTAL fails to preprocess *fs* due to running out of the disk space. As a result, we cannot obtain any execution time of CRYSTAL on *fs*. Figure 8 presents the execution time excluding the preprocessing time. DUALSIM fails due to OOT because of its slow in-memory enumeration. In contrast, SEED and CRYSTAL generally fail due to OOS, which is caused by the large number of intermediate results. Specifically, SEED fails to process P_5 on *eu*, since the reducers run out of memory space. CRYSTAL fails on P_5 on *uk* due to OOT. The other failure cases of SEED and CRYSTAL are caused by running out of the disk space. LIGHT is the only algorithm that finishes all the 42 test cases (7 pattern graphs $\times$ 6 data graphs), whereas DUALSIM, SEED and CRYSTAL fail in 16, 8 and 12 cases respectively. As shown in Figure 8, LIGHT can outperform DUALSIM by up to three orders of magnitude. Furthermore, LIGHT running on a single machine is up to two orders of magnitude faster than both SEED and CRYSTAL running on a cluster of 12 machines.

IX. CONCLUSION

In this paper, we propose an efficient parallel subgraph enumeration algorithm LIGHT for a single machine. To reduce redundant computation, we propose the lazy materialization and the minimum set cover based candidate set computation. Moreover, we parallelize LIGHT with SIMD and SMT to fully utilize the parallel computation capabilities in modern CPUs. Detailed experimental results show that LIGHT outperforms the state-of-the-art algorithms by orders of magnitude.

X. ACKNOWLEDGMENTS

This work was partly supported by grants 16206414 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft.

REFERENCES

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *TODS*, 2017.
[2] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *SIGPLAN*, 2013.
[3] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, 2013.
[4] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, 2008.
[5] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.
[6] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. In *SIAM Journal on Computing*, 1985.
[7] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Annual International Conference on Research in Computational Molecular Biology*, 2007.
[8] M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *ACM-SIAM symposium on Discrete algorithm*, 2006.
[9] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, 2013.
[10] S. R. Kairam, D. J. Wang, and J. Leskovec. The life and death of online groups: Predicting group growth and longevity. In *ACM international conference on Web search and data mining*, 2012.
[11] 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 *SIGMOD*, 2016.
[12] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. In *PVLDB*, 2015.
[13] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. In *PVLDB*, 2016.
[14] D. Lemire, L. Boytsov, and N. Kurz. Simd compression and the intersection of sorted integers. In *Software: Practice and Experience*, 2016.
[15] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. In *TODS*, 2014.
[16] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. *arXiv*, 2018.
[17] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *PODS*, 2012.
[18] H.-M. Park, S.-H. Myaeng, and U. Kang. Pte: Enumerating trillion triangles on distributed systems. In *SIGKDD*, 2016.
[19] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. In *PVLDB*, 2017.
[20] V. N. Rao and V. Kumar. Parallel depth first search. part i. implementation. In *International Journal of Parallel Programming*, 1987.
[21] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD*, 2014.
[22] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, 2009.
[23] J. Ugander, L. Backstrom, and J. Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *WWW*, 2013.
[24] J. R. Ullmann. An algorithm for subgraph isomorphism. In *JACM*, 1976.
[25] T. L. Veldhuizen. Leapfrog triejoin: a simple, worst-case optimal join algorithm. In *ICDT*, 2014.
[26] S. Wernicke. Efficient detection of network motifs. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2006.