Parallel Subgraph Listing in a Large-Scale Graph

Yingxia Shao[♯] Bin Cui[♯] Lei Chen[§] Lin Ma[♯] Junjie Yao[♯] Ning Xu[♯]
[♯]Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
[§]Department of Computer Science and Engineering, HKUST
[♯]{simon0227, bin.cui, malin1993ml, jjyao, ning.xu}@pku.edu.cn
[§]leichen@cse.ust.hk

ABSTRACT

Subgraph listing is a fundamental operation to many graph and network analyses. The problem itself is computationally expensive and is well-studied in centralized processing algorithms. However, the centralized solutions cannot scale well to large graphs. Recently, several parallel approaches are introduced to handle the large graphs. Unfortunately, these parallel approaches still rely on the expensive join operations, thus cannot achieve high performance.

In this paper, we design a novel parallel subgraph listing framework, named PSgL. The PSgL iteratively enumerates subgraph instances and solves the subgraph listing in a divide-and-conquer fashion. The framework completely relies on the graph traversal, and avoids the explicit join operation. Moreover, in order to improve its performance, we propose several solutions to balance the workload and reduce the size of intermediate results. Specially, we prove the problem of partial subgraph instance distribution for workload balance is NP-hard, and carefully design a set of heuristic strategies. To further reduce the enormous intermediate results, we introduce three independent mechanisms, which are automorphism breaking of the pattern graph, initial pattern vertex selection based on a cost model, and a pruning method based on a light-weight index.

We have implemented the prototype of PSgL, and run comprehensive experiments of various graph listing operations on diverse large graphs. The experiments clearly demonstrate that PSgL is robust and can achieve performance gain over the state-of-the-art solutions up to 90%.

Categories and Subject Descriptors

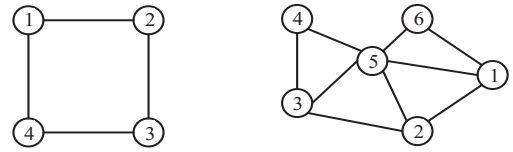
H.3.3 [**Information Search and Retrieval**]: Search Process; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

Keywords

Graph; Graph Algorithm; Subgraph Listing; Graph Enumeration

SIGMOD/PODS'14, June 22 - 27 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2588557.



(a) Pattern Graph G_p (b) Data Graph G_d

Figure 1: Subgraph Listing Sample

1. INTRODUCTION

Subgraph listing [6] is an operation to find all the occurrences of a pattern graph in a data graph. Figure 1 shows a square pattern graph G_p, and a data graph G_d. The goal of the operation is to enumerate all the subgraph instances in G_d that are isomorphic with G_p, such as $\square_{1235}, \square_{1256}, \square_{2345}$. This is a fundamental operation in the frequent subgraph mining, network processing and motif discovering in bioinformics [22]. Moreover, the emergence of social network analyses also requires subgraph listing. For example, mining frequent subgraphs can reveal the information cascade patterns in real life [19] and counting triangles helps compute the clustering coefficient of a social network [26].

Though it is very useful, the subgraph listing is computationally challenging [1]. The size of the result set is often exponential to the number of vertices in the pattern graph. Most of the algorithms enumerate[1] the subgraph instances one by one [6, 13], and cannot handle large graphs. Several stream-based approaches [31, 4] were proposed to tackle the large graphs. However, these solutions can only output the approximate occurrence number and the isomorphic subgraph instances are not available. Recently, parallel solutions [1, 24] were proposed for the subgraph listing problem on MapReduce [8]. The approach [1] follows the parallel query processing paradigm. That is, it decomposes the pattern graph into small ones (edges), extracts the subgraphs for each small pattern graph and joins the intermediate results finally. The performance of the parallel solution is limited by the expensive join operator. Meanwhile, the imbalance distribution of data graph or intermediate results also hinders the performance of these solutions.

In this paper, we design a novel parallel subgraph listing framework, named PSgL. PSgL is originated from the fact that each subgraph instance is independent. In other words, these subgraph instances can be enumerated concurrently without confliction. Therefore, from the view of these subgraph instances, the subgraph listing is an embarrassingly parallel problem [10]. We solve the problem in a parallel and divide-and-conquer fashion, to improve the performance of subgraph listing in large graphs. In PSgL, the subgraph listing operation is iteratively divided into partial subgraph listings by the graph traversal, and each worker expands the *par-*

[1]we use "enumerate" and "list" interchangeably.

tial subgraph instance, meanwhile, communicates with each other through the distribution strategy. The process is proceeded until all the subgraph instances are found. The whole execution is based on the graph traversal, thus avoiding the costly join operation.

Referring to the example in Figure 1, the existing approaches [1, 24] will generate a path of length four through join operation and verify the equality of two ends for the square pattern graph. However, in PSgL, after creating the path of length three by graph traversal, it can verify the partial subgraph instance fast through checking the neighborhood of one end without generating paths of length four. This implies that PSgL will not produce more complicated intermediate results with the help of graph traversal.

However, there still exist two challenges to efficiently perform subgraph listing in PSgL. One is the workload balance problem. To a parallel computing framework, the imbalance is the curse for the performance improvement. In PSgL, a totally different set of partial subgraph instances is processed in every iteration, and the workload characteristic varies sharply between iterations. The other is the enormous partial subgraph instance problem. Since the result set exponentially increases when the size of the pattern graph grows, the partial subgraph instances will be even more than the result set due to the invalid ones. These enormous valid and invalid partial subgraph instances heavily affect the computation, communication and memory cost.

To address the first challenge, we prove the hardness of the problem and carefully design several distribution strategies for the PSgL when it distributes the partial subgraph instances in each iteration. They are random distribution strategy, roulette wheel distribution strategy and workload-aware distribution strategy. For the workload-aware distribution strategy, we clearly model the cost of expanding a partial subgraph instance and propose a simple but effective heuristic rule to achieve a good workload balance.

For the second one, we first break the automorphism of the pattern graph, thus guaranteeing each subgraph instance is found exactly once and reducing the number of duplicated partial subgraph instances during the runtime. We next present a cost model to select a good initial pattern vertex which helps improve the overall performance. At last, we design a light-weight edge index to rule out the invalid partial subgraph instances as early as possible.

We have implemented the prototype of PSgL, and run comprehensive experiments on large-scale graphs. The results demonstrate that PSgL outperforms the state-of-the-art parallel solutions.

The contributions of our work are summarized as follows:

- We propose an efficient parallel subgraph listing framework, PSgL, which can well handle large-scale graphs.
- We introduce a cost model for the subgraph listing in PSgL. The cost model can help design a workload-aware distribution strategy and select a good initial pattern vertex.
- We propose a simple but effective workload-aware distribution strategy, which facilitates PSgL to achieve good workload balance.
- We introduce a deterministic rule to select the "best" initial pattern vertex for cycles and cliques.
- We design a light-weight edge index, which can filter invalid partial subgraph instances efficiently.

The rest of this paper is organized as follows: we introduce the related work and preliminaries in Section 2 and 3. Section 4 elaborates the PSgL framework followed by presenting our optimization techniques in Section 5. The implementation details are discussed in Section 6. We show the experimental results in Section 7. Finally, we conclude the work in Section 8.

2. RELATED WORK

Subgraph listing is a basic operation to the graph and complex network analyses. Many research works have been conducted on this problem, which can be classified into three categories: centralized algorithms, streaming algorithms and parallel algorithms. Here we briefly review these works.

Centralized algorithms. Most solutions previously introduced are the centralized processing approaches. N. Chiba et al. [6] proposed a simple edge-searching based strategy, which derives algorithms for listing triangle, quadrangle, clique in $\mathcal{O}(\alpha(G)m)$ time, where m is the number of edges in G and $\alpha(G)$ is the arboricity of G. Authors [28, 13] improved the performance of centralized algorithms by ensuring that each subgraph is found only once. However, these centralized algorithms enumerate subgraphs one by one, and cannot handle large-scale graphs. Some efficient centralized approaches [16, 7] for large-scale graphs have been proposed as well, but these solutions only focus on the special pattern graph, triangle.

Streaming algorithms. In order to handle the massive graphs, several approaches [4, 31, 5] were introduced based on the data stream model [23]. These methods apply the sampling, probability and statistic techniques to count the approximated occurrence number of the pattern graphs. They are efficient to process large-scale graphs, however, they cannot list all the isomorphic subgraph instances. Furthermore, the works based on the approximated results may lead to inaccurate conclusion.

Parallel algorithms. Another promising approach to handle subgraph listing on large-scale graphs is the parallel computing. Afrati et al. [1] introduced a single map-reduce round method to enumerate the subgraph instances. The solution distributes the edges of a data graph with efficient mapping schemes, and joins them at each reducer. T. Plantenga [24] proposed a SGIA-MR algorithm within MapReduce framework. The algorithm finds the results based on a pre-defined edge join order in several iterations and exhibits excellent scalability. However, they still need the expensive join operations, and generate a large amount of invalid intermediate results. Meanwhile, the imbalanced distribution of data or large intermediate results heavily affects the performance. Furthermore, several parallel solutions for counting triangles on a large-scale graph were also proposed [26, 18, 12].

Another close research field is subgraph matching. The subgraph matching runs on the property graph, where each vertex has attributes. The core of the work is to improve the response time. In [14, 30, 32], several attribute-based indexes are introduced, which can efficiently prune the invalid intermediate results to reduce the latency for each query. Recently, Z. Sun [25] proposed a parallel approach for subgraph matching. The approach decomposes the original pattern graph into several STwigs, solves STwigs in a certain order and finally assembles the STwigs' results together through join operation. This approach also focuses on graph query processing and reduces the latency of query answering on a large graph. The subgraph listing can be viewed as a special case of subgraph matching problem, where all the vertices have the same attributes. Because of this difference, existing attribute-based indexes fail to work on subgraph listing and the size of results of subgraph listing increases exponentially compared with the one of subgraph matching. Furthermore, the subgraph listing is an off-line analytic task and focuses on reducing the total execution time.

Though a lot of works have been dedicated to the subgraph listing operation, the problem of efficient subgraph listing on large scale graphs is still open. In this paper, we design the PSgL framework totally relying on the graph traversal, and endeavor to seek an efficient solution for the subgraph listing operation.

3. PRELIMINARIES

Problem & Notations. Subgraph listing is a classic operation in graph computing. It enumerates all the instances of a pattern graph in a data graph, and both graphs have no labels on vertices and edges. In this paper, we design the PSgL framework to efficiently execute the operation on undirected graphs ***in parallel***. A graph is denoted by $G = (V, E)$, where V and E are the sets of vertices and edges. For each vertex $v \in V$, $N(v)$ denotes the neighborhood of v, and $deg(v)$ is the degree of v, which equals to $|N(v)|$. We distinguish the pattern graph and data graph by subscript p and d.

Now we proceed to present several key conceptions for the analysis of PSgL.

Power-law Graph is a graph whose degree distribution follows a power law. That is, the probability of a vertex having a degree d is given by:

$$p(d) \propto d^{-\gamma},$$

where the parameter γ is a positive constant that controls the skewness of the degree distribution. A lower γ indicates that more vertices are high degree, i.e., more skewed.

Random Graph is a graph that is generated by some random processes. The classic random model is the Erdős-Rényi model [9]. The degree distribution of the ER random graph follows the poisson distribution, and most of the vertices have the degrees around the average.

Ordered Graph is an undirected graph with manual assignment of partial order for the vertices. In this paper, the data graph G_d is ordered by following two rules:

1. for any v_d,$u_d \in V_d$, if $deg(v_d) < deg(u_d)$, then $v_d < u_d$;
2. if $deg(v_d)$=$deg(u_d)$ and v_d has a smaller vertex id, then v_d<u_d.

For a vertex v_d in the ordered graph, we use n_b to denote the number of neighbors who have smaller rank, and n_s to represent the counter part. The property of n_b and n_s is described below.

PROPERTY 1. *The distribution of n_b is more skewed than the original degree distribution, while n_s is more balanced.*

Here we give a brief explanation. Assume in the original data graph, the probability that a vertex has a degree d is $p(d)$. Then, for a vertex v_d, whose degree is d, n_s and n_b are

$$n_b = d \times \sum_{d_i<d} p(d_i), \quad n_s = d \times (1 - \sum_{d_i<d} p(d_i)) \tag{1}$$

From Equation 1, it is easy to figure out that higher d derives higher n_b and lower d leads to lower n_b. This implies the distribution of n_b will be more polarized compared with the original. In contrast, the distribution of n_s will be more concentrated to the average and more balanced. Taking a power-law graph, WebGoogle, as an example, the original degree distribution has $\gamma = 1.66$. After ordering it, γ is 1.54 for the n_b distribution while it has $\gamma = 3.97$ for the n_s distribution.

Partial subgraph instance is a data structure that records the mapping between G_p and G_d, denoted by G_{psi}. G_{psi} consists of $|V_p|$ vertex mapping pairs, where each pair is represented by <v_p,v_d> (v_d=$map(v_p)$). Thus, assume the vertices of G_p are numbered from 1 to $|V_p|$, we can also simply state G_{psi} as {map(1), map(2), ..., map($|V_p|$)}. In Figure 1, for example, the G_{psi} for the original G_p is {?, ?, ?, ?}, where "?" means the v_p has no mapped v_d, while the G_{psi} for $\square_{1256}$ is {1, 2, 5, 6}. In addition, we use *subgraph instance* to represent the found subgraph in data graph, such as $\square_{1235}$, $\square_{1256}$, $\square_{2345}$. The original pattern graph combining a partial subgraph instance forms a *partial pattern graph*, G_{pp}, as illustrated in Figure 2(a). The vertex color of G_{pp} will be explained in Section 4.3.

Automorphism of a Pattern Graph. The automorphism of a pattern graph $G_p = (V_p, E_p)$ is a permutation σ of the vertex set V_p, such that the pair of vertices (u_p, v_p) forms an edge if and only if the pair $(\sigma(u_p), \sigma(v_p))$ also forms an edge. That is, it is a graph isomorphism from G_p to itself. The automorphism causes the same subgraph instance found multiple times. For example, without any preprocessing, the $\square_{2345}$ can be found eight times by the pattern graph in Figure 1, because there exist eight valid permutations that make the square pattern graph isomorphism to itself. In order to guarantee that each subgraph instance is found exactly once, we need to eliminate the automorphism of a pattern graph. We name this procedure as *automorphism breaking*, and the approach is described in Section 5.2.1.

4. PARALLEL SUBGRAPH LISTING FRAMEWORK

The **P**arallel **S**ub**g**raph **L**isting (PSgL) framework follows the state-of-the-art graph processing paradigm [21], which applies the vertex-centric programming model and Bulk Synchronous Parallel [27]. The data graph is randomly distributed among the workers' memory. PSgL iteratively expands the partial subgraph instances by data vertices in parallel, until all the subgraph instances are found. In each iteration, the newly created partial subgraph instances are sent across the workers according to the distribution strategy. The entire enumeration process relies on the partial subgraph instance and its independence property.

In the following subsections, we discuss the properties of partial subgraph instance first. Then we elaborate the PSgL framework based on the partial subgraph instance and the core partial subgraph instance expansion algorithm. At last, we analyze the cost of PSgL.

4.1 Independence Property

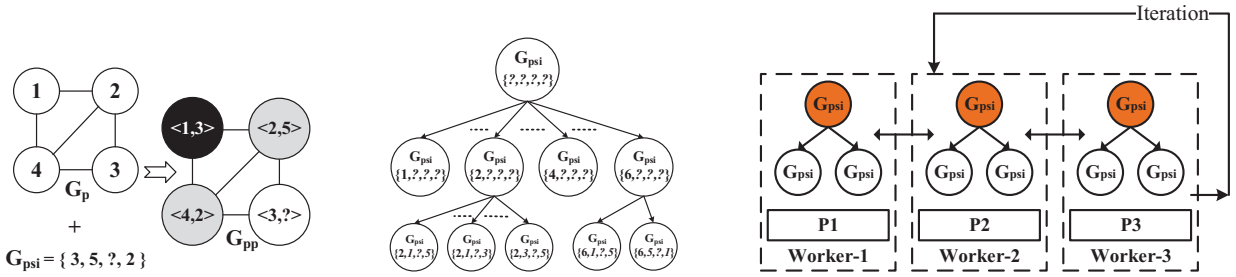
Partial subgraph instance is a key intermediate result for the enumeration process. All the partial subgraph instances during an enumeration form a tree, called G_{psi} tree. A node in the tree corresponds to a G_{psi} and the children of a node are derived from expanding one mapped data vertex in the node (Section 4.3). The root of the tree is the G_{psi} obtained from the input pattern graph. Figure 2(b) illustrates (a part of) the G_{psi} tree for square pattern graph listing on data graph in Figure 1. {?,?,?,?} is the root. By expanding the mapped data vertex 6 in {6,?,?,?}, it generates two new G_{psi}s, {6,1,?,5} and {6,5,?,1}. Therefore, {6,1,?,5} and {6,5,?,1} are the children of {6,?,?,?} in the tree.

A single G_{psi} encodes a set of subgraph instances in its subtree. G_{psi}s at the same level of the tree are a complete representation of the whole result set. From top to down in a G_{psi} tree, the original enumeration task is divided into more fine-grained subtasks (subtrees). Moreover, we notice that the partial subgraph instance is independent from each other except the ones in its generation path. In other words, after the partial subgraph instance is generated, it can be processed independently without being aware of other ones. For example, {2,1,?,3} and {6,5,?,1} can be simultaneously computed on data vertex 1 and 5 respectively. We name this property as the *independence property*.

The tree hierarchy and independence property allow the subgraph listing problem to be solved in a **divide-and-conquer** style. We can first divide the problem into partial subgraph listing and then conquer the partial subgraph instance and generate new ones in parallel, the process is repeated until all results are returned.

4.2 PSgL Framework

Our proposed PSgL is in charge of constructing the G_{psi} tree for a pattern graph in parallel. It consists of two distinct phases,



(a) G_{pp} (b) G_{psi} Tree (c) Iterative Process in PSgL Framework

Figure 2: Partial Pattern Graph, Partial Subgraph Instance Tree and PSgL Framework

Algorithm 1 Expand a Partial Subgraph Instance

Input: partial subgraph instance G_{psi}, pattern graph G_p
1: get the current expanding v_p and v_d from G_{psi}
2: $candList \leftarrow NULL$, $color(v_p) \leftarrow$ BLACK
3: /* explore v_p's neighbor */
4: **foreach** v_p' in $N(v_p)$ **do**
5: **if** processNeighbor(v_p',v_d,$candList$)=**false then**
6: **return**;
7: **end if**
8: **end foreach**
9: **if** isComplete(G_{psi})=**true then**
10: **print** subgraph instance
11: **else**
12: generate new G_{psi}s by combining candidates in $candList$ and distribute them based on distribution strategy in Algorithm 3.
13: **end if**

i.e., initialization and expansion, and both are concentrated on the partial subgraph instance.

Initialization phase. In this phase, each data vertex creates a G_{psi} which only contains a vertex mapping pair of the data vertex and the selected initial pattern vertex (Section 5.2.2). These G_{psi}s form the initial set of partial subgraph instances and correspond to a one-level G_{psi} tree as the start.

Expansion phase. This is the core procedure in the PSgL framework and the whole G_{psi} tree will be constructed after this phase. Figure 2(c) shows the high-level execution flow of expansion phase. It consists of several iterations. In an iteration, where the divide-and-conquer happens, each G_{psi} is expanded independently on a certain worker, and generates more fine-grained ones. These new G_{psi}s are sent to the next worker according to the distribution strategy, if they are not the complete subgraph instances. The iteration ends when the previously received G_{psi}s are processed and the new ones are received by the corresponding workers. The phase will not finish until there is no G_{psi} in the framework.

Please note that PSgL may not guarantee that each G_{psi} is expanded in the same pace, which means that the G_{psi} tree doesn't grow level by level. It only guarantees that all the G_{psi}s in PSgL cover the results which have not been found. Nevertheless, PSgL is also suitable for the case where the tree grows one level in each iteration.

4.3 Partial Subgraph Instance Expansion

The partial subgraph instance expansion algorithm generates more fine-grained G_{psi}s and makes the G_{psi} tree grow. It explores the partial pattern graph, which contains three kinds of vertices: BLACK, GRAY and WHITE. Figure 2(a) shows a G_{pp} during the expansion.

BLACK vertex is the one which has been expanded. The neighbors of a BLACK vertex must consist of BLACKs and GRAYs.

GRAY vertex has a mapped data vertex, but it has not been expanded. It must be adjacent to at least one BLACK vertex. Moreover, the GRAY vertices represent the expanding candidate set of this partial subgraph instance for the following process.

Algorithm 2 Process a Neighbor of Pattern Vertex

Input: v_p', v_d, $candList$
Output: updated $candList$
1: **if** color(v_p') = GRAY **then**
2: **if** the $map(v_p')$ is not in $N(v_d)$ **then**
3: **return false**
4: **end if**
5: **else if** color(v_p') = WHITE **then**
6: /* details are described in Algorithm 5*/
7: $candidate \leftarrow$ getCandidateSet(v_p')
8: **if** $candidate$ = NULL **then**
9: **return false**
10: **end if**
11: add the $candidate$ into $candList$
12: **end if**{BLACK v_p' is omitted.}
13: **return true**

WHITE vertex is the one which hasn't been mapped to any data vertex.

The expansion algorithm changes one GRAY vertex, which is specified by the previous iteration, into BLACK, and also makes its neighbor become GRAY if it is WHITE. The expansion procedure is presented in Algorithm 1. It first obtains the expanding pattern vertex v_p from G_{psi}, sets v_p in BLACK (Lines 1-2), and then processes v_p through neighbor exploration on the pattern graph (Lines 4-8). In the end (Lines 9-13), it prints the found subgraph instance, if the G_{psi} is completed. Otherwise, it sets the WHITE neighbors in GRAY, and creates new G_{psi}s by combining neighbors' candidates with pruning invalid combinations via the rules in Section 5.2.3. For each new G_{psi}, it selects a GRAY vertex according to the distribution strategy as the next expanding vertex and sends the new G_{psi} to the corresponding worker.

Algorithm 2 illustrates the details for processing a neighbor of v_p. It processes the neighbor according to its color. If it is GRAY, we need to verify whether the edge exists in the data graph or not. It retrieves the candidates for the WHITE vertex from the neighbors of data vertex v_d. BLACK vertices are just omitted because they have been expanded previously and the corresponding edge must exist in the data graph.

4.4 Cost Analysis

In this subsection, we analyze the performance of PSgL and reveal the metrics related to its cost. Because the partial subgraph instance is the minimal computing unit in the framework, let's first consider the cost of processing a single partial subgraph instance in the expansion phase, denoted as $load(G_{psi})$. From Algorithm 1, the $load(G_{psi})$ consists of communication and computation costs. The communication cost is caused by distributing the new G_{psi}s, while the generating process itself causes the significant computation cost. The asynchronous communication model [21] facilitates PSgL to do the communication and computation concurrently, so the $load(G_{psi})$ is determined by the heavier one. On account of

the problem itself is computation intensive, the $load(G_{psi})$ is approximate to the computation cost, in most iterations.

The computation cost includes the cost of verifying the GRAY neighbors ($cost_g$) and the cost of generating new G_{psi}s by retrieving candidates for the WHITE neighbors ($cost_w$). We use c_e to represent the cost of generating a single G_{psi} and $f(v_p)$ to denote the number of new G_{psi}s through expanding v_p. The $load(G_{psi})$ can be calculated as

$$load(G_{psi}) = cost_g + cost_w = cost_g + c_e \times f(v_p). \quad (2)$$

Based on the $load(G_{psi})$, the cost of a worker can be represented as $L = \sum_{i=1}^{N} load(G_{psi})$, where N is the number of G_{psi} processed by the worker.

Assume that the subgraph listing task runs on a system with K workers, and the task finishes in S iterations. During the i^{th} iteration, the worker k processes N_{ki} G_{psi}s. Then we have the total cost of the task T as

$$T = \sum_{i=1}^{S} \max_{1 \le k \le K} \{L_{ki}\} = \sum_{i=1}^{S} \max_{1 \le k \le K} \{\sum_{j=1}^{N_{ki}} load(G_{psi})_j\} \quad (3)$$

To achieve high performance in PSgL, it is required to minimize the total cost T. From the above equation, we can figure out there are three metrics affecting the overall cost.

First is the number of iterations S. In general, the fewer number of iterations achieves better performance, since there is some overhead between iterations, like synchronization. The following theorem shows the bound of S for a pattern graph.

THEOREM 1. *Given a pattern graph $G_p = (V_p, E_p)$, if the G_{psi} tree grows level by level, then S is restricted by $|MVC| \le S \le |V_p| - 1$, where $|MVC|$ is the size of the minimum vertex cover of G_p.*

PROOF. Because the G_{psi} tree grows level by level, in each iteration, only one v_p is expanded. Thus PSgL must visit at least $|MVC|$ vertices before every edge in G_p is explored. But it will not exceed $|V_p| - 1$ after all the edges are explored. In conclusion, S is limited by $|MVC| \le S \le |V_p| - 1$. □

In PSgL, not all the partial subgraph instances have the same expansion path because of the distribution strategy, and hence we do not explicitly optimize the S. Its effects are considered during the initial pattern vertex selection.

The second issue is the workload balance which is implied by the max function. As a classic parallel computing framework, it can be damaged by the imbalance. So we design and compare several distribution strategies in Section 5.1 to investigate which one provides better performance.

The last is the number of G_{psi}s. As above equations indicate that the performance of worker in an iteration is related to N_{ki} and $f(v_p)$, we need the expansion algorithm produce as small size of G_{psi} as it can. This will reduce the computation cost and communication cost at the same time. We propose several techniques to achieve this goal in Section 5.2.

5. OPTIMIZATION TECHNIQUES

In this section, we introduce the optimization techniques to improve the performance of PSgL. These techniques are classified into two categories. One is the partial subgraph instance distribution strategy for the workload balance, and the other is for reducing the enormous partial subgraph instances.

5.1 Distribution Strategy

As Section 4.4 discussed, the distribution of G_{psi}s should make the workload balance among the workers in an iteration. In general, a good data graph partition will help to achieve this goal. However, due to the generality of the input pattern graph and the variable characteristic of workload of G_{psi}s between iterations, it is difficult to design a one-size-fit-all graph partition.

In PSgL, the data graph is simply random partitioned, and the G_{psi}s are distributed online without fixed expansion path to avoid the workload imbalance. For a single G_{psi}, there are several choices when distributing it. Illustrated in Figure 2(a), all the GRAY vertices are the candidates for the next expansion. It is possible to achieve workload balance by selecting a good destination for each G_{psi}.

The cost of a G_{psi} has been modeled in Equation 2. Here we define the increased workload w_i of worker i, if a G_{psi} is sent to the worker i. The w_i can be formulated as

$$w_i = \begin{cases} \min_{\text{GRAY } v_p} \{load(G_{psi})_{v_p}\} & map(v_p) \text{ belongs to } i; \\ +\infty & otherwise. \end{cases} \quad (4)$$

We proceed to define the **partial subgraph instance distribution problem** as below:

DEFINITION 1. *There are N partial subgraph instances to be processed by K workers, and the cost of a partial subgraph instance processed by worker i is defined in Equation 4. We use W_i to denote the total cost for the partial subgraph instances processed by worker i. Then the goal is to find a distribution strategy for the N partial subgraph instances to achieve*

$$\min\{\max_{1 \le i \le K} \{W_i\}\}$$

The following theorem describes the hardness of the partial subgraph instance distribution problem.

THEOREM 2. *The partial subgraph instance distribution problem is NP-hard.*

PROOF. Minimum Makespan Scheduling [17], which is a well known NP-hard problem, can be easily reduced to the partial subgraph instance distribution problem. A typical Minimum Makespan Scheduling problem on unrelated machines can be stated as "There are m parallel machines and n independent jobs. Each job is to be assigned to one of the machines. The processing of job j on machine i requires time P_{ij} . The objective is to find a schedule that minimizes the makespan". So if j^{th} job maps to the j^{th} G_{psi} and the cost of job j processed by machine i maps to w_{jk}, then the Minimum Makespan Scheduling problem is reduced to the partial subgraph instance distribution problem. □

In order to solve this NP-hard problem efficiently, we propose the following heuristic rules.

5.1.1 Workload Aware Distribution Strategy

For the partial subgraph instance distribution problem, all the partial subgraph instances are generated online, so we need an online solution. The classical heuristic rule for this situation is selecting worker j for the i^{th} G_{psi} which has minimal overall workload,

$$\arg\min_{j} \{W_j + w_{ij}\}$$

In [17], the authors proved that this greedy rule is tightly bounded by $K \times OPT$. However, this greedy solution is likely to obtain a local optimum as it always tries to balance the cost first when a new item comes in.

We refine the heuristic rule by reducing the penalty part and it will increase the opportunity to jump out the local optimum. The general form of the heuristic rule can be

$$\arg\min_{j} \{W_j^{\alpha} + w_{ij}\}, 0 \le \alpha \le 1,$$

where W_j^{α} stands for the penalty part which restricts the objective to be balanced. When $\alpha = 1$, it is the original heuristic rule.

When $\alpha = 0$, it implies that each time the rule chooses worker j where the G_{psi} incurs the least increased workload. But this rule is more likely to be imbalanced, though the total workload would be minimized.

Based on the above observations, we choose $\alpha = 0.5$ and make a trade off between the balance and minimal workload. To some extend, the rule with $\alpha = 0.5$ can avoid local optimum more likely than $\alpha = 1$, while it achieves more balance than $\alpha = 0$. Besides, the following theorem guarantees that in the worst case, the performance of $\alpha = 0.5$ is still bounded by $K \times OPT$.

THEOREM 3. *For the partial subgraph instance distribution problem, the overall cost achieved by the heuristic rule with $\alpha = 0.5$ is no K times worse than the OPT.*

PROOF. Let $s_i = \min_{1 \le j \le K} w_{ij}$ which indicates the minimal cost of i^{th} G_{psi} processed across all the K workers and each w_{ij} is positive integer in our problem.

Then let the lower bound of the cost for n G_{psi} as

$$g(n) = \sum_{i=1}^{n} s_i$$

It is easy to infer that $OPT \ge \frac{1}{K} \times g(N)$. Let $f(n)$ represent the overall cost obtained by the greedy algorithm. Each time when i^{th} G_{psi} is distributed to worker k, the following inequations are held:

$$W_k^{\alpha} + w_{ik} \le W_j^{\alpha} + w_{ij}, 1 \le j \le K \tag{5}$$

We next prove the theorem by inducing the task id.

(1) $i = 1, f(1) = s_1 \le g(1)$

(2) $f(i) = \max(f(i-1), W_k + w_{ik})$

(2.a) $f(i-1) \ge W_k + w_{ik}$, then $f(i) = f(i-1) \le g(i-1) \le g(i-1) + s_i \le g(i)$

(2.b) $f(i-1) \le W_k + w_{ik}$, then $f(i) = W_k + w_{ik}$, and Equation 5 makes sure that $W_k^{0.5} + w_{ik} \le W_j^{0.5} + s_i \le f(i-1)^{0.5} + s_i$.

First, we can get the inequations: $f(i) - w_{ik} \le f(i-1) \le f(i)$

Then,

$$\begin{aligned}
W_k^{0.5} + w_{ik} &\le f(i-1)^{0.5} + s_i \\
[f(i) - w_{ik}]^{0.5} + w_{ik} &\le f(i-1)^{0.5} + s_i \\
(w_{ik} - s_i)^2 &\le f(i-1) + [f(i) - w_{ik}] \\
&\quad - 2f(i-1)^{0.5} \times [f(i) - w_{ik}]^{0.5} \\
&\le f(i-1) + [f(i) - w_{ik}] \\
&\quad - 2[f(i) - w_{ik}]^{0.5} \times [f(i) - w_{ik}]^{0.5} \\
f(i) &\le f(i-1) + s_i + (w_{ik} - s_i) - (w_{ik} - s_i)^2 \\
&\le f(i-1) + s_i \le g(i-1) + s_i \le g(i)
\end{aligned}$$

At last, we obtain $f(N) \le g(N) \le K \times OPT$. □

In the following, we show how to estimate $load(G_{psi})$ in practice. Because $cost_g$ is only caused by neighborhood search, which can be done efficiently by a bitmap index, we can estimate the cost of a G_{psi} as $f(v)$. The value of $f(v_p)$ is limited to $[0, \binom{deg(v_d)}{w_{v_p}}]$ range, here w_{v_p} is the number of WHITE neighbors around v_p. Subsequently, we use the upper bound of $f(v)$ to estimate itself, thus guaranteeing the estimated value and the accurate $f(v)$ have the same order. So $load(G_{psi}) \simeq \binom{deg(v_d)}{w_{v_p}}$, if it expands v_p for the G_{psi}. The pseudo-code of the workload-aware distribution strategy is presented in Algorithm 3. The time complexity of the algorithm can be $\mathcal{O}(|V_p|)$ by only calculating the approximate value of $\binom{deg(v_d)}{w_{v_p}}$.

Algorithm 3 Distribution Strategy Summary

```
1:  Roulette Wheel Distribution Strategy
2:  foreach GRAY v_p in G_psi do
3:      p_{v_p} ← calculate through Equation 6.
4:  end foreach
5:  randnum ← Random(0,1)
6:  foreach GRAY v_p in G_psi do
7:      if randnum ≤ p_{v_p} then
8:          return v_p;
9:      end if
10:     randnum ← randnum − p_{v_p}
11: end foreach
```

```
1:  Workload Aware Distribution Strategy
2:  W_j indicates the total workload for worker j
3:  i ← current G_psi id
4:  k ← 0 // k records the selected worker id
5:  v_k records the selected expanding v_p
6:  w_ik records the increased workload for worker k
7:  foreach GRAY v_p in G_psi do
8:      j ← worker id of map(v_p)
9:      if W_k^α + w_ik > W_j^α + C(deg(v_d), w_{v_p}) then
10:         k ← j, w_ik ← C(deg(v_d), w_{v_p}), v_k ← v_p
11:     end if
12: end foreach
13: W_k ← W_k + w_ik
14: return v_k
```

5.1.2 Naive Distribution Strategies

Here we introduce two other distribution strategies for the partial subgraph instance distribution problem. They are random distribution strategy and roulette wheel distribution strategy.

Random distribution strategy is randomly choosing a GRAY vertex for a G_{psi}. It will balance the number of G_{psi} processed by a worker. However, the different G_{psi}s will have various effects to the workload and the cost is still imbalance among workers. Clearly, this strategy has its own merit, i.e., it is simple and has the minimal overhead.

Roulette wheel distribution strategy considers the degree information of the data graph when distributing a partial subgraph instance. Since $load(G_{psi})$ is approximated to $f(v_p)$, it implies the larger degree of a data vertex results in higher cost for a G_{psi}. Thus the following heuristic rule can be derived.

HEURISTIC 1. *The data vertex with larger degree should expand less G_{psi}.*

The rule indicates G_{psi} is preferred to be expanded by a data vertex with small degree. Considering the balance problem, we propose a distribution strategy based on the roulette wheel selection [2].

The roulette wheel distribution strategy chooses GRAY vertex k in G_{psi} with a probability p_k. p_k is defined in the following equation:

$$p_k = \frac{\prod_{j=1, j \ne k}^{n} deg(v_{dj})}{\sum_{i=1}^{n} \prod_{j=1, j \ne i}^{n} deg(v_{dj})} \tag{6}$$

where n is the number of GRAYs in G_{psi}, v_{dj} is the data vertex mapping to GRAY vertex v_{pj}.

The probability indicates that G_{psi} has a higher chance to be expanded by a data vertex with smaller degree. In order to avoid the imbalance, it is still possible that some G_{psi}s are distributed to the high degree data vertices. The strategy can achieve better balance than the random strategy. However, due to the probability p_k is unaware of previous G_{psi}'s distribution, it may cause some data vertices with small degree overloaded.

As the probability of each v_p can be calculated in $\mathcal{O}(1)$ with proper preprocess, the roulette wheel distribution is a linear strategy with time complexity $\mathcal{O}(|V_p|)$. The procedure is described in Algorithm 3.

5.2 Partial Subgraph Instance Reduction

The enormous partial subgraph instances consume a lot of memory resources, and introduce expensive generation and communication cost as well. It is important to reduce the number of partial subgraph instances, in order to improve PSgL's performance. However, the size of partial subgraph instances is related to many factors, including the structure of the pattern graph, structure of the data graph, the initial pattern vertex, distribution strategies and so on. It is tough to design one solution for such a complex problem. We propose three independent mechanisms from three aspects to reduce the size of partial subgraph instances. They are automorphism breaking of the pattern graph, initial pattern vertex selection based on a cost model, and a pruning method based on a lightweight index.

5.2.1 Automorphism Breaking of the Pattern Graph

As mentioned in Section 3, automorphism breaking of a pattern graph guarantees each subgraph instance is found exactly once, it reduces the duplicated partial subgraph instances. However, the classical graph labeling method [11] for breaking automorphism cannot handle our problem, because the labeling has nothing to do with the data graph, which even has no label.

Since the data graph has been ordered by its degree sequence, we also assign a partial order set for the pattern graph. The partial order set not only breaks the automorphism of the pattern graph, but also can be used to prune partial subgraph instances. Because the graph automorphism problem is still unknown whether it has a polynomial time algorithm or it is NPC problem [20], breaking it has the same difficulty. Here we introduce an approach by iteratively assigning a partial order on the pattern graph until the graph is not automorphism any more.

The main idea, in each iteration, is to pick an equivalent vertex group, where each vertex can be mapped to others in a certain automorphism of the pattern graph, and eliminate a member from the group by assigning a partial order, which sets the lowest rank to the eliminated member compared to the remained ones. We use the DFS to find the equivalent vertex group in a pattern graph, and it has been demonstrated that DFS can detect automorphism of a graph with up to 100 vertices in seconds [13].

However, there are several partial order sets to break the automorphism of a graph. Because PSgL follows the graph traversal, the partial order on edges can be immediately used during the exploration, and prune invalid partial subgraph instances early. Moreover, as the order between vertices who are not connected can only be used after exploration, we apply the following heuristic rule to select a good partial order set.

HEURISTIC 2. *In each iteration, the algorithm selects the equivalent vertex group which contains vertices with higher degree to break.*

5.2.2 Initial Pattern Vertex Selection

Initial pattern vertex is the one where the algorithm starts to traverse. Fixing an initial pattern vertex prevents producing duplicated subgraph instances. However, different initial pattern vertices generate partial subgraph instances in different size and have various influences on the performance of a pattern graph. We design a cost model-based initial pattern vertex selection for the general pattern graph. Besides, we derive a deterministic rule based on the model for two special pattern graphs: cycles and cliques.

Algorithm 4 Cost Estimation for a Pattern Vertex

Input: v_p, G_p
1: $estimatedCost \leftarrow 0$
2: $l \leftarrow 0, n \leftarrow |V_d|, G_{pp} \leftarrow G_p$ /*l means the number of iteration.*/
3: mark v_p GRAY in G_{pp}.
4: $queue \leftarrow (G_{pp}, n, l)$
5: **while** $queue$ is not empty **do**
6: $(G_{pp}, n, l) \leftarrow queue.front(); queue.pop()$
7: $estimatedCost \leftarrow estimatedCost + cost(G_{pp}, n, l)$
8: **if** G_{pp} is expandable **then**
9: **foreach** $GRAY$ v_p in G_{pp} **do**
10: expand and generate new $(G'_{pp}, n', l+1)$
11: **if** $(G'_{pp}, n', l+1)$ exists **then**
12: update the existed $(G'_{pp}, n', l+1)$
13: **else**
14: $queue.push((G'_{pp}, n', l+1))$
15: **end if**
16: **end foreach**
17: **end if**
18: **end while**
19: **return** $estimatedCost$

General pattern graph. The optimal initial pattern vertex should be the one which leads to the minimal cost. We design a cost model to estimate the cost of a certain initial pattern vertex, and select the vertex with minimal estimated cost as the "best" initial pattern vertex.

For a general pattern graph, the initial pattern vertex selection framework enumerates all the pattern vertices, and for each one, calculates its cost by traversing from the selected vertex along with estimating the cost for each generated partial pattern graph (Algorithm 4). The overall estimated cost for an initial pattern vertex equals to the sum of the cost of all the partial pattern graphs. The time complexity of selection framework is $\mathcal{O}(|V_p| \times |E_p|)$.

The core portion of the framework is the model to estimate the cost of a partial pattern graph, $cost(G_{pp}, n, l)$. In the selection framework, we assume that the random distribution strategy is used. When there are n G_{psi}s for G_{pp} to be expanded, the cost of these G_{psi}s can be estimated as:

$$cost(G_{pp}) = n \times (cost_g + \frac{1}{C}\sum_{i=1}^{C} c_e \times f(v_{pi})),$$

where C is the number of GRAY vertices in G_{pp}, and $cost_g$ is similar for different GRAY vertices.

As it does not know the data vertex that v_p maps to, the approach in Section 5.1 fails to estimate $f(v_p)$. But, it is easy to obtain the degree distribution $p(d)$ of the data graph by sampling or traversing, we can estimate $f(v_p)$ with the following equation:

$$f(v_p) \simeq \sum_{d=deg(v_p)}^{d_{max}} p(d) \times \binom{d}{w_{v_p}}$$

Based on the selection framework and cost estimation model, we can choose a good initial pattern vertex for the general pattern graph.

Cycles and cliques. The initial pattern vertex selection framework indicates the following theorem.

THEOREM 4. *Under the initial pattern vertex selection framework, the best initial pattern vertex is the one which derives a traversal order minimizing the total number of partial subgraph instances.*

PROOF. Assuming the simulation for a pattern vertex terminates in S iterations and, in the l^{th} iteration, there exist T_l different partial pattern graphs, each has n_{lt} corresponding G_{psi}s. The total

number of G_{psi} in iteration l is n_l. As the random distribution strategy is applied, so $n_l = n_{lt} \times T_l$. Then the total estimated cost for the pattern vertex is:

$$T_e = \sum_{l=0}^{S} \sum_{t=1}^{T_l} n_{lt} \times (cost_g + \frac{1}{C} \sum_{i=1}^{C} c_e \times f(v_{pi}))$$

Next, we define g_l as the *average expanding coefficient* at iteration l. The g_l can be represented as

$$g_l = \begin{cases} 1 & l = 0 \\ \frac{1}{T_l} \frac{1}{C} \sum_{t=1}^{T_l} \sum_{i=1}^{C} f(v_{pi}) & l \neq 0. \end{cases}$$

Now, T_e can be represented by g_l

$$T_e = \sum_{l=0}^{S} (n_l \times cost_g + c_e \times n_l \times g_l) \propto \sum_{l=0}^{S} n_l \times g_l \quad (7)$$

Here we ignore $cost_g$ with the same reason in Section 5.1.

The term $n_l \times g_l$ is the number of newly generated G_{psi} in iteration l. So from Equation 7, we can conclude, under our selection framework, a traversal order minimizing the total number of partial subgraph instances results into minimal estimated cost T_e, which implies the corresponding initial pattern vertex is the best. □

For the cycles and cliques, after breaking the automorphism, there must be a vertex who has the lowest rank, because the first equivalent vertex group contains all the pattern vertexes. Then we have a ***deterministic rule***, that is the vertex with the lowest rank is the best initial pattern vertex for the cycles and cliques, and the following theorem shows the correctness.

THEOREM 5. *After breaking the automorphism of cycles and cliques, the vertex v_{l_r} with the lowest rank is the best initial pattern vertex for any ordered data graph.*

PROOF (sketch). I) first step, $l = 1$.

$$g_1 = f(v_p) = \frac{1}{|V_d|} \sum_{i=1}^{|V_d|} \binom{deg(v_{di})}{w_{v_p}} \quad \because C = 1, T_1 = 1$$

For cliques and cycles, w_{v_p}=$|V_p| - 1$ and w_{v_p}=2, respectively. On an ordered graph with the partial order pruning, $deg(v_{di})$=n_s or n_b. Because $\sum_{i=1}^{|V_d|} n_s (or\ n_b)$=$|E_d|$, and Property 1 holds, we can easily infer that v_{l_r} has the minimal g_1.

II) remained steps, $l > 1$. For all the cliques we have $g_l = 1$, because of w_{v_p}=0. For the cycles, due to w_{v_p}=1, all the g_ls are linear to the degree and result in $g_i \simeq g_j, j > i > 1$ (*sophisticated analyses are omitted*).

At last, Equation 7 can derive

$$T_e \propto \sum_{l=0}^{S} n_l \times g_l = n_0 \times \sum_{l=0}^{S} \prod_{i=0}^{i=l} g_i \quad (8)$$

Based on the characteristics of g_l and Equation 8, we can work out that v_{l_r}, which has minimal g_1, leads to the minimal number of partial subgraph instances, for the cycles and cliques. According to Theorem 4, v_{l_r} is the best initial pattern vertex. □

Though, Theorem 5 points out v_{l_r} is the best initial pattern vertex, the improvement still depends on the original degree distribution of the data graph. For the power-law graph, where the distributions of n_b and n_s can be totally different[2], v_{l_r} can enhance the performance significantly. Given a random graph, where the n_b and n_s are similar after ordering, PSgL may not benefit a lot from v_{l_r}.

[2]refer to the example in Section 3.

Algorithm 5 Get Candidate Set

Input: a WHITE neighbor v_p' of v_p
Output: candidates $cand$

```
1: foreach v_d' in N(v_d) do
2:   /*pruning rule 1.*/
3:   if deg(v_d') < deg(v_p') and partial order restriction then
4:     continue
5:   end if
6:   /*pruning rule 2.*/
7:   valid ← true
8:   foreach v_p'' in N(v_p') do
9:     if color(v_p'') = GRAY and checkEdgeExistence(v_d', map(v_p'')) =
       false then
10:      valid ← false
11:      break
12:    end if
13:  end foreach
14:  if valid = false then
15:    continue
16:  end if
17:  add u_d' into cand
18: end foreach
```

5.2.3 *Pruning Invalid Partial Subgraph Instance*

The aforementioned two techniques reduce the size of partial subgraph instances off-line. However, during the runtime, many invalid partial subgraph instances, which do not lead to find other subgraph instances, are generated. The later they are pruned, the more resources they consume.

In order to reduce the number of invalid partial subgraph instances, the quality of the candidate set of WHITE vertices needs to be improved. This can be done by efficient filtering rules. However, without label information, most existing pruning techniques [14, 32] fail in this context. The only information we can use is the graph structure and the partial order. First, the degree information filters the partial subgraph instance, if $deg(v_d) < deg(v_p)$. Second, it is the neighbor connectivity. When retrieving candidates for a WHITE neighbor of the expanding pattern vertex v_p, it should guarantee that the edge between the neighbor and v_p's GRAY neighbor exists in the data graph. In Figure 2(a), we need to check edge (3,4) when expanding vertex 2. At last, during the expansion, the partial order must be consistent between the pattern graph and data graph.

Since the data graph is stored in distributed memory, it is expensive to check an edge's existence remotely. So we design a light-weight edge index in PSgL for fast checking the existence of an edge in data graph. It is an inexact index which is built on the bloom filter [3], and indexes the ends of an edge. The index can be built in $\mathcal{O}(m)$ time and consumes a small memory footprint. Moreover, the precision of the index is adjustable and the successive iteration only needs to verify a small portion of partial subgraph instances.

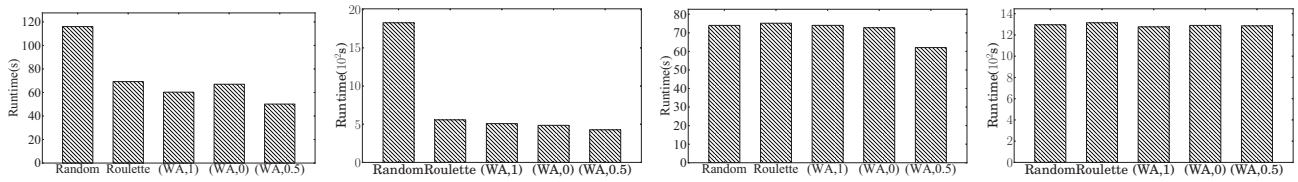
With the help of above pruning rules, it can generate high-quality candidate sets. Algorithm 5 illustrates the procedure of candidate generation for a WHITE vertex. It first uses the degree constrains and partial order restriction to filter the invalid candidates, and then checks all the v_p''s GRAY neighbors through the light-weight index.

6. IMPLEMENTATION DETAILS

In this section, we present the implementation details of PSgL. The prototype of PSgL is written in Java on Giraph[3], which is an open-source Pregel first released by Facebook.

The design of initialization phase and expansion phase in PSgL follows the vertex-centric model, so both phases are integrated into

[3]https://github.com/apache/giraph



(a) PG_2 on WebGoogle (b) PG_2 on WikiTalk (c) PG_2 on UsPatent (d) PG_4 on LiveJournal

Figure 3: Performance of Various Distribution Strategies

a single vertex program on Giraph. The first superstep (iteration) of the vertex program is responsible to execute the initialization phase. The following supersteps, each data vertex processes the incoming G_{psi}s by Algorithm 1. The messages communicated among workers not only include G_{psi}, but also encode the status information, such as the next expanding pattern vertex, the colors of pattern vertices and the progress of G_{psi}. This is the basic implementation of PSgL, and the G_{psi} tree grows one level in each iteration. Besides the basic vertex program, PSgL also requires several kinds of shared data, i.e., pattern graph, initial pattern vertex, light-weight index, and degree statistics. Considering these data that are all small enough to be stored on a single node, (i.e., the edge index of Twitter dataset only costs 2GB.), in current version of PSgL, each worker maintains a copy of them. Moreover, these shared data are static, so we compute them off-line once and load them before running the vertex program through preApplication() API in WorkerContext object.

Furthermore, the distributor is a specific module of PSgL for supporting various distribution strategies. Since the distributor selects the next expanding pattern vertex for a G_{psi} and is shared by all the local data vertices, it can be initialized in preApplication() as well. The two naive distribution strategies only rely on the static shared information, which are easy to implement locally. The workload-aware distribution strategy ideally needs the dynamical global information of each worker's workload W_i. However, in the parallel execution setting, it is expensive to maintain such a global view. Instead, during the distribution, each worker only maintains a local view of the entire workload distribution. Thus the update of W_k can be done fast without communication and synchronization. In practice, since a partition usually contains a moderate size of vertices, it is possible to make a good distributing decision according to the local information.

7. EXPERIMENTS

In this section, we evaluate the performance of PSgL. The following subsection describes the environment, datasets and pattern graphs. We experimentally demonstrate the effectiveness of optimization techniques in Section 7.2, 7.3 and 7.4. Section 7.5 presents the comparison results on various pattern graphs. At last, we evaluate the scalability of PSgL on large graphs and the number of workers in Section 7.6 and 7.7 respectively.

7.1 Experimental Setup

All the experiments were conducted on a cluster with 28 nodes, where each node is equipped with an AMD Opteron 4180 2.6GHz CPU, 48GB memory and a 10TB disk RAID.

Pattern graphs. We use five different pattern graphs, PG_1 to PG_5, illustrated in Figure 4. The partial orders obtained from automorphism breaking are listed below the pattern graph.

Graph datasets. We use six real-world graphs and one synthetic graph in our experiments. All the real-world graphs are undirected ones created from the original release by adding reciprocal

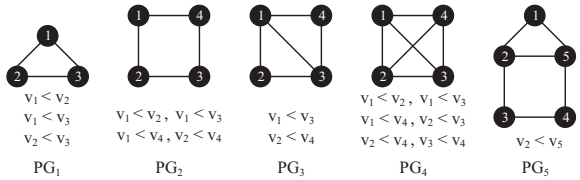


Figure 4: Pattern Graphs

edge and eliminating loops and isolated nodes. Except Wikipedia, which can be downloaded from KONECT, other real-world graphs are available on SNAP. The random graph is generated by NetworkX following the Erdős-Rényi model. Table 1 summarizes the meta data of these graphs.

	Web-Google	Wiki-Talk	Us-Patent	Live-Journal	Wiki-pedia	Twitter	Rand-Graph
$\|V\|$	0.9M	2.4M	3.8M	4.8M	26M	42M	4M
$\|E\|$	8.6M	9.3M	33M	85M	543M	1,202M	80M

Table 1: Meta Data of Graphs

During the experiments, we only output the occurrences of the pattern graph. But we generate the found subgraph instances and can store them if necessary. Each experiment is ran five times and we showed the average performance without including the loading time in the paper. The reported runtime is the real job execution time which includes both communication and computation costs.

7.2 Effects of Distribution Strategies

We evaluate five distribution variants, i.e., random distribution strategy, roulette wheel distribution strategy and workload-aware distribution strategy with three different parameters, which are named (WA,0), (WA,0.5), (WA,1). Take (WA,0) as an example, it means the workload aware distribution strategy which has $\alpha = 0$.

Since the distribution strategy balances the workload for each iteration, the effect of distribution strategies varies according to the different characteristics of an iteration. Here we show the results of PG_2, where middle ($l > 1$) iterations generate new partial subgraph instances, and PG_4, where only the first ($l = 1$) iteration generates partial subgraph instances, as representatives. We profiled the experiments and verified the effectiveness of the (WA,0.5) strategy.

From Figures 3(a), 3(b) and 3(c), we can see that the strategy (WA,0.5) can achieve about 77% improvement on the WikiTalk against the random distribution strategy, when running PG_2. Compared with other three strategies, it can still have around 11% to 23% improvement. There are similar results on the WebGoogle. On the UsPatent, the improvement is not as significant as previous two graphs. This is because the degree distributions of WebGoogle and WikiTalk are seriously power-law skewed, which have $\gamma = 1.66$ and $\gamma = 1.09$ respectively, while the power-law parameter γ of UsPatent is 3.13. It reveals that the strategy (WA,0.5) is obviously benefit the graphs with skewed degree distribution, when the pattern graph generates new partial subgraph instances in the middle iteration.

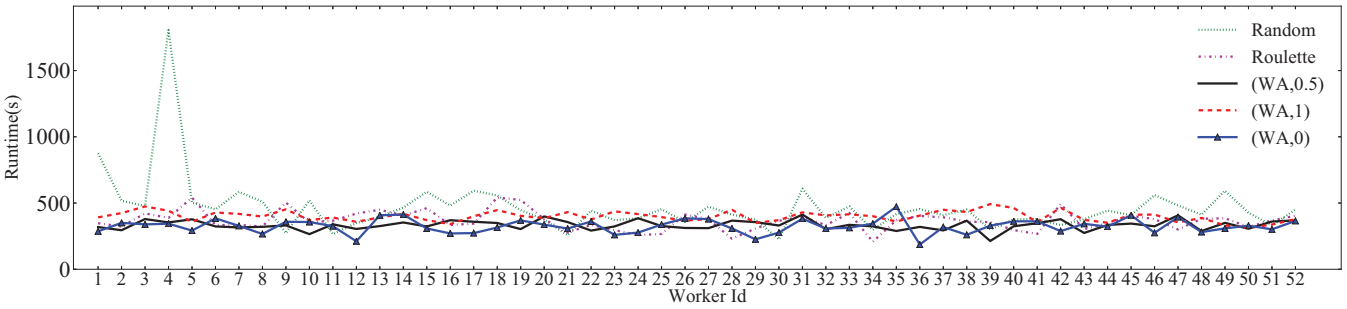
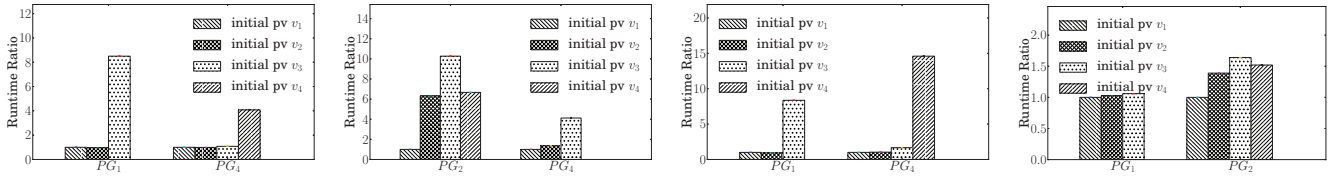


Figure 5: Each Worker's Performance on WikiTalk with PG_2



(a) LiveJournal (b) WikiTalk (c) WebGoogle (d) RandGraph

Figure 6: Influences of the Initial Pattern Vertex on Various Data Graphs

In Figure 3(d), all the five strategies have similar performance when running PG_4 on the LiveJournal. For the clique pattern graph, it only generates the partial subgraph instances in the first iteration which is affected by the initial pattern vertex, and the following iterations are for the verification, which is an operation with constant cost. So the other distribution strategies can also obtain good performance. This implies the performance of a distribution strategy is related to the pattern graph as well.

Figure 5 shows the detailed performance of each worker when running PG_2 on WikiTalk. It illustrates that the strategy (WA,0.5) achieves balance while minimizing the cost of the slowest worker. Though, the strategy (WA,1.0) achieves similar balance, it is stuck into the local optimum and cannot minimize the cost of the slowest worker. While strategy (WA,0) can guarantee most workers have smaller cost, but it cannot achieve better balance, the worker 35 performs much slower than the others. In addition, the slowest worker is different between random distribution strategy and roulette wheel distribution strategy. It is because the vertices with higher degree cause the imbalance in random distribution strategy, while the ones with smaller degree having too much workload cause the imbalance in roulette wheel distribution strategy. These phenomena are consistent with previous discussions in Section 5.1.

7.3 Importance of the Initial Pattern Vertex

Here we report the results on the power-law graph and random graph with running PG_1, PG_2 and PG_4, which have a deterministic rule to identify the good initial pattern vertex according to the cost model. This will clearly reason the importance of selecting a good initial pattern vertex.

Figures 6(a), 6(b) and 6(c) show the performance of different initial pattern vertices on the power-law graph. The real runtime of each initial pattern vertex is normalized to the runtime of the best initial pattern vertex for each pattern graph, so we present the runtime ratio in figures. For the clarity of the figure, we did not present the runtime ratio which exceeds 100 times over the best initial pattern vertex on WikiTalk. From the figures, we notice that for PG_1 on LiveJournal, it is about 8.5 times slower by selecting the highest rank v_3 as the initial pattern vertex than selecting the lowest rank v_1. Note that v_2 has the similar performance to v_1, because there exists an edge (v_2, v_3) with order $<$ for the v_2 in PG_1. The gap is even larger on WikiTalk, which reaches about 285 times. The clique pattern graph PG_4 has the similar results. The gap is 4 times on LiveJournal, while it is 106.4 times on WikiTalk. On the web graph, WebGoogle, the initial pattern vertex of PG_1 and PG_4, has the similar effects as on the social graph. The improvements are 8.4 and 14.6 times, for PG_1 and PG_4, respectively. Therefore it is necessary to choose a good initial pattern vertex when enumerating a certain pattern graph on real-world graphs. The deterministic rule in Theorem 5 is effective for this task.

However, Figure 6(d) shows all the three initial pattern vertices in PG_1 have the similar performance on the random graph, and for PG_2, the gap is only about 1.6 times between the slowest and fastest initial pattern vertexes. It indicates that the influence of initial pattern vertex is less significant on random graph than the one on power-law graph, for the cycles and cliques. This is consistent with previous analysis in Section 5.2.2.

7.4 Efficiency of the Light-Weight Edge Index

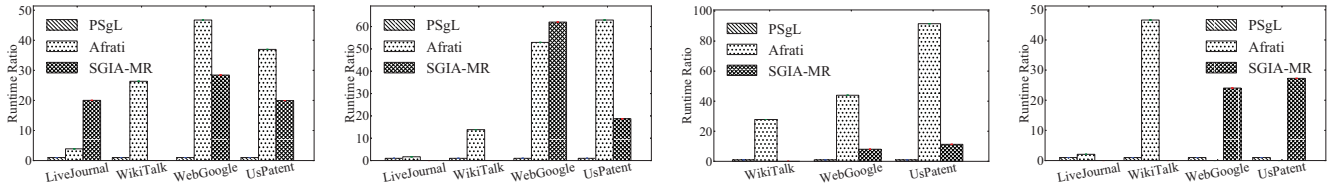
Next we present the results on the efficiency of the light-weight edge index. With the help of the light-weight edge index, PSgL can filter invalid partial subgraph instances early, and saves the communication and memory costs.

Data Graph	PG	G_{psi}# w/ index	G_{psi}# w/o index	Pruning Ratio
LiveJournal	$PG_1(v_1)$	2.86×10^8	6.81×10^8	58.01%
	$PG_4(v_1)$	9.93×10^9	OOM	Unknown
UsPatent	$PG_5(v_1)$	2.26×10^7	3.17×10^8	92.87%
	$PG_5(v_3, v_4)$	7.38×10^9	2.04×10^{10}	63.89%

Table 2: Pruning Ratio of the Edge Index. $PG_i(v_j)$ stands for the number of G_{psi} is counted during the expansion of v_j for the PG_i.

Table 2 only lists the online statistics for executing the pattern graph PG_1, PG_4, PG_5 on LiveJournal and UsPatent, as representatives due to the space constraint, as the other pattern graphs give similar results. We notice that the pruning ratio can be as high as 92.87% during the expansion of v_1 for the PG_5 on UsPatent. Without the edge index, the 92.87% invalid partial subgraph instances will cause heavy communication and memory consumption. On LiveJournal, when running PG_4 starting from v_1 without the edge index, the task fails with OutOfMemory[4] (OOM for short) problem because of the enormous invalid partial subgraph instances. Fur-

[4]using the terminology in Java to denote the error.



(a) PG_1 (b) PG_2 (c) PG_3 (d) PG_4

Figure 7: Runtime Ratio among PSgL, Afrati and SGIA-MR

thermore, in the different iterations, the index has different pruning ratio. It has small pruning ratio in the later iterations during which the size of partial subgraph instances is closer to the result set.

In summary, if there exists the invalid partial subgraph instances, the edge index can prune them efficiently and reduce the memory and communication overhead.

7.5 Performance on Various Pattern Graphs

Now we evaluate the performance of PSgL on the real datasets with various pattern graphs, PG_1 to PG_5. All the optimization techniques discussed in Section 5 are used. We compare PSgL with the MapReduce solutions (i.e., Afrati [1] and SGIA-MR [24]). For one input pair of pattern graph and data graph, the cost of each solution is normalized to the cost of PSgL. The normalized cost is called runtime ratio and is presented in figures. A runtime ratio x for a solution $\mathcal{A}$ means the performance of $\mathcal{A}$ is x times slower than PSgL's performance. For the clarity of the figures, the ratios exceed 100 times are not visualized. In addition, as the MapReduce solutions cannot be finished in four hours for PG_5, we did not show the results in figures either, and the results of PG_3 on LiveJournal are omit for the same reason.

Figure 7 lists the runtime ratio between three solutions. We can easily see from the figure, that PSgL significantly outperforms the MapReduce solutions on the WikiTalk, WebGoogle and UsPatent. On average, PSgL can achieve performance gains over the MapReduce soultions around 90%. Especially, when executing PG_4 on UsPatent, the runtime ratio between PSgL and Afrati can be 225 (not visualized). The join operation makes the reducer operate slowly. Besides the graph traversal advantage in PSgL, the online distribution strategy helps PSgL avoid the serious imbalance and achieve a good performance across various settings. In contrast, the MapReduce solutions have varieties of performance across the different datasets, and the two surpass each other interleaved. This is because the distribution strategy of intermediate results in these solutions follows a fixed scheme and the degree of the skewness for each MapReduce solution changes sharply with different graphs. For example, SGIA-MR spends 4213 seconds for PG_1 on WikiTalk and Afrati finishes in only 190 seconds. However, on WebGoogle, Afrati is about 1.6 times slower than SGIA-MR.

Furthermore, though, the speed up of PSgL on LiveJournal is around 2 to 4 times across different pattern graphs, the absolute saved time is significant. For example, when running PG_2, PSgL finishes in 4302 seconds, while Afrati consumes 7291 seconds. For PG_5, even though, we assume the MapReduce solutions finish in four hours, the speed up of PSgL is around 3 to 14 times on different data graphs.

7.6 Scalability on Large Graphs

To evaluate the scalability of our approach on large graphs, we further conduct experiments on two large datasets, Twitter and Wikipedia, and also compare with GraphChi [18] and PowerGraph [12]. GraphChi and PowerGraph are two state-of-the-art graph computing systems on a single node and in parallel, respectively. For both methods, the latest C++ versions are used in the experiments.

Data Graph	Pattern Graph	Afrati	Power-Graph(C++)	Graph-Chi(C++)	PSgL
Twitter	PG_1	432[5]min	2min	54min	**12.5**min
Wikipedia	PG_1	871s	36s	861s	**125**s

Table 3: Triangle Listing on Large Graphs

Meanwhile, we show the robustness of our proposal by comparing with a one-hop index based solution on PowerGraph.

Table 3 shows the comparison results of triangle counting on Wikipedia and Twitter, since triangle (PG_1) is a very typical pattern graph and attracts more attention in social network analysis. We can see that PSgL achieves performance gains over the MapReduce solution up to 97% on Twitter and 86% on Wikipedia. It surpasses GraphChi as well. However, compared with PowerGraph, PSgL is about 4 to 6 times slower. One reason is that PowerGraph is a heavily optimized graph-parallel execution engine [29]. Furthermore, the triangle counting operation on PowerGraph is well optimized via a one-hop neighborhood index maintained by hopscotch hashing [15].

By using the one-hop index technique, it is insufficient to enable PowerGraph to solve a general pattern graph efficiently. We extend the graph traversal based solution in PSgL to the PowerGraph. Unlike the original PSgL, we manually choose a traversal order for the general pattern graph, and let PowerGraph solve the subgraph listing based on that order and prune the intermediate results via the one-hop index. The traversal order is denoted by like "A->B->C", which means the algorithm first visits vertex "A", then "B" and "C" on the pattern graph. Furthermore, we eliminate the automorphism of the pattern graph to guarantee each result is found once as well.

Data Graph	Pattern Graph	Traversal Order	Afrati	Power-Graph	PSgL
WikiTalk	PG_2	1->2->3->4	4402s	**48**s	318s
WikiTalk	PG_3	2->3->4->1	13743s	**100**s	494s
WikiTalk	PG_3	1->2->3->4	13743s	OOM	**494**s
WikiTalk	PG_4	1->2->3->4	1785s	127s	**38**s
LiveJournal	PG_4	1->2->3->4	2749s	OOM	**1330**s
WebGoogle	PG_5	1->2->3->4 ->5	>4h	OOM	**4232**s

Table 4: General Pattern Graph Listing Comparison

Table 4 illustrates the performances of PSgL and PowerGraph on general pattern graphs. For the simple pattern graph PG_2, similar to the triangle PG_1, PowerGraph can obtain better performance with the help of one-hop index. While pattern graphs become complicated, PowerGraph degrades because of the enormous invalid intermediate results, and cannot handle general pattern graphs even the data graph is small. For example, when executing PG_4 on WikiTalk, PowerGraph is 3.3 times slower than PSgL. The result is even worse on LiveJournal, the task is failed with the OutOfMemory (OOM) problem. This is because, without the global edge index, the algorithm is unable to use the connectivity except the one-hop link to prune the invalid intermediate results, thus burdening the memory and communication overhead. Moreover, unlike PSgL

[5]The performance is cited from [26].

where the distribution strategy dynamically chooses the traversal order for each G_{psi}, the fixed traversal order cannot achieve a balanced distribution of intermediate results. When running PG_5 on PowerGraph, the imbalanced distribution leads to OOM on some nodes. Another important observation is, similar to the initial pattern vertex selection in PSgL, the different fixed traversal orders heavily affect the performance and it is difficult for a non-expert to figure out a good traversal order for the general pattern graphs. For instance, when executing PG_3 with traversal order "2->3->4->1", it leads to better performance while the performance degrades significantly with the traversal order "1->2->3->4". Because "1->2->3->4" makes the algorithm generate a large set of initial intermediate results and causes the OOM issue.

In summary, PSgL is a subgraph listing framework to support the general pattern graphs by addressing above problems via the workload-aware distribution strategy, a cost model-based initial pattern vertex selection method and the light weight edge index. The experimental results show PSgL's scalability and robustness with respect to various pattern graph types and datasets.

7.7 Scalability on the Number of Workers

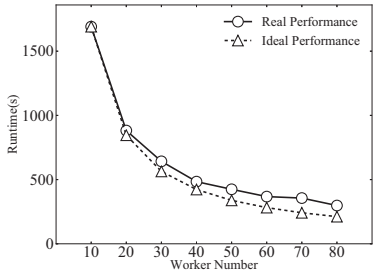


Figure 8: Performance vs. Worker Number

Finally we demonstrate that PSgL has a graceful scalability with the increasing number of workers. Figure 8 shows the performance of running PG_2 on the WikiTalk with the number of workers raising from 10 to 80. We notice that the real performance curve is approximate to the ideal curve, which assumes the performance is linear to the worker number. The runtime is decreased closely linear with respect to the worker number. For example, it is about 1691 seconds for running PG_2 with 10 workers, while the cost is reduced to 845 seconds when the workers is doubled. As the number of workers goes up, the improvement decreases slightly.

8. CONCLUSION

In this paper, we have proposed an efficient parallel solution, called PSgL, to address the subgraph listing problem on large-scale graphs. PSgL is a parallel iterative subgraph listing framework, which is graph friendly and designed based on the basic graph operation. Moreover, we introduced several optimization techniques to balance the workload and reduce the size of intermediate results, which can further enhance the performance of PSgL. Through the comprehensive experimental study, we demonstrated that PSgL outperforms the state-of-the-art solutions in general.

ACKNOWLEDGMENTS

The research is supported by the National Natural Science Foundation of China under Grant No. 61272155. Lei Chen's work is supported in part by the Hong Kong RGC/NSFC Project N_HKUST 637/13, National Grand Fundamental Research 973 Program of China under Grant 2012-CB316200, Microsoft Research Asia Gift Grant and Google Faculty Award 2013.

9. REFERENCES

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. ICDE, 2013.
[2] T. Bäck. Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press, 1996.
[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 1970.
[4] I. Bordino, D. Donato, A. Gionis, and S. Leonardi. Mining large networks with subgraph counting. ICDM, 2008.
[5] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. PODS, 2006.
[6] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. SIAM J. Comput., 1985.
[7] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. KDD, 2011.
[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI, 2004.
[9] P. Erdős and A. Rényi. On random graphs. I. Publ. Math. Debrecen, 1959.
[10] I. Foster. Designing and building parallel programs: Concepts and tools for parallel software engineering. AWL Co., Inc., 1995.
[11] J. A. Gallian. A dynamic survey of graph labeling. Electron. J. Combin., 2000.
[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.
[13] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. RECOMB, 2007.
[14] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. SIGMOD, 2008.
[15] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. DISC, 2008.
[16] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. SIGMOD, 2013.
[17] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. J. ACM, 1977.
[18] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. OSDI, 2012.
[19] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. PAKDD, 2006.
[20] A. Lubiw. Some np-complete problems similar to graph isomorphism. SIAM J. Comput., 1981.
[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. SIGMOD, 2010.
[22] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. Science, 2002.
[23] S. Muthukrishnan. Data streams: algorithms and applications. SODA, 2003.
[24] T. Plantenga. Inexact subgraph isomorphism in mapreduce. J. Parallel Distrib. Comput., 2013.
[25] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. PVLDB, 2012.
[26] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. WWW, 2011.
[27] L. G. Valiant. A bridging model for parallel computation. Commun. ACM, 1990.
[28] S. Wernicke. Efficient detection of network motifs. Trans. Comput. Biol. Bioinformatics, 2006.
[29] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: a resilient distributed graph system on spark. GRADES Workshop In SIGMOD, 2013.
[30] P. Zhao and J. Han. On graph query optimization in large networks. PVLDB, 2010.
[31] Z. Zhao, M. Khan, V. S. A. Kumar, and M. V. Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. ICPP, 2010.
[32] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu. Treespan: efficiently computing similarity all-matching. SIGMOD, 2012.