Scalable Subgraph Enumeration in MapReduce

Longbin Lai[§], Lu Qin[‡§], Xuemin Lin[§♮], and Lijun Chang[§]

[§] The University of New South Wales, Australia
[‡]Centre for QCIS, University of Technology, Sydney, Australia
[♮]East China Normal University, China

[§]{llai,lxue,ljchang}@cse.unsw.edu.au; [‡]lu.qin@uts.edu.au

ABSTRACT

Subgraph enumeration, which aims to find all the subgraphs of a large data graph that are isomorphic to a given pattern graph, is a fundamental graph problem with a wide range of applications. However, existing sequential algorithms for subgraph enumeration fall short in handling large graphs due to the involvement of computationally intensive subgraph isomorphism operations. Thus, some recent researches focus on solving the problem using MapReduce. Nevertheless, exiting MapReduce approaches are not scalable to handle very large graphs since they either produce a huge number of partial results or consume a large amount of memory. Motivated by this, in this paper, we propose a new algorithm TwinTwigJoin based on a left-deep-join framework in MapReduce, in which the basic join unit is a TwinTwig (an edge or two incident edges of a node). We show that in the Erdös-Rényi random-graph model, TwinTwigJoin is instance optimal in the left-deep-join framework under reasonable assumptions, and we devise an algorithm to compute the optimal join plan. Three optimization strategies are explored to improve our algorithm. Furthermore, we discuss how our approach can be adapted in the power-law random-graph model.We conduct extensive performance studies in several real graphs, one of which contains billions of edges. Our approach significantly outperforms existing solutions in all tests.

1. INTRODUCTION

In this paper, we study subgraph enumeration, which is a fundamental problem in graph analysis. Given an undirected, unlabeled data graph G and a pattern graph P, subgraph enumeration aims to find all subgraph instances of G that are isomorphic to P. Subgraph enumeration is widely used in a lot of applications. For example, subgraph enumeration is used for network motif computing [27, 5] to facilitate the design of large networks from biochemistry, neurobiology, ecology, and bioinformatics. Subgraph enumeration is utilized to compute the graphlet kernels for large graph comparison [28, 30] and property generalization for biological networks [26]. Subgraph enumeration is considered as a key operation for the synthesis of target structures in chemistry [31]. Subgraph enumeration can also be adopted to illustrate the evolution of social networks [20] and to discover the information trend in recommendation networks [24]. In addition, as a special case of subgraph enumeration, triangle enumeration is a preliminary operation in cluster coefficient calculation [40] and community detection [39].

Proceedings of the VLDB Endowment, Vol. 8, No. 10
Copyright 2015 VLDB Endowment 2150-8097/15/06.

Motivation. Despite a large variety of applications, enumerating subgraphs in a big data graph is very challenging. The reasons are twofold. First, subgraph enumeration is computationally intensive, since determining whether a data graph contains a subgraph that is isomorphic to a given pattern graph, known as subgraph isomorphism, is NP-hard. Second, the lack of label information makes it hard to filter infeasible partial answers in early stages, rendering a large number of partial results, whose size can be much larger than the size of the data graph and the final results. Due to these challenges, existing sequential algorithms for subgraph enumeration [8, 16] are not scalable to handle big graphs. Some other studies try to find approximate solutions [5, 15, 43] to reduce the computational cost, however, they only estimate the count of the matched subgraphs rather than locate all the subgraph instances.

MapReduce [11], as one of the most popular parallel computing paradigms for big data processing, has been widely used in both industry and academia. MapReduce embodies the advantages of high scalability, reliability, and fault-tolerance. Its easy-to-use programming model allows developers to develop highly scalable data-driven algorithms in a distributed environment. Therefore, MapReduce has recently been used for subgraph enumeration in big graphs to pursue both scalability and efficiency. In the literature, two existing approaches focus on subgraph enumeration using MapReduce, namely, edge-based join [29] and multiway join [1].

In edge-based join [29], the pattern graph is decomposed into an ordered list of edges, and the algorithm proceeds in multiple MapReduce rounds where each round grows one edge using the join operation. Edge-based join is inefficient, as joining one edge in each round cannot fully make use of the structural information, which may render numerous partial results.In multiway join [1], only one MapReduce round is needed for subgraph enumeration. In the algorithm, each edge is duplicated in multiple machines such that each machine can enumerate the subgraphs independently and no match is missed. However, multiway join usually encounters serious scalability problems by keeping almost the whole graph in the memory of each machine when the pattern graph is complex.

Considering the drawbacks of edge-based join and multiway join, in this paper, we propose a new approach for subgraph enumeration in MapReduce. We introduce a left-deep-join framework that generalizes the edge-based join to allow the right join argument to be a star (a tree of depth 1) rather than a single edge in each round. However, joining a star is sometimes inefficient as well. Thus, we propose the TwinTwigJoin, which uses a simple TwinTwig (an edge or two incident edges of a node) as the right join argument in each round. TwinTwigJoin, as a tradeoff between edge-based join and star-based join, has several advantages. First, based on a well-defined cost model and the Erdös Rényi random-graph model, we show that under reasonable assumptions, TwinTwigJoin can ensure instance optimality in the left-deep-join framework. Second, the simple structure of a TwinTwig makes it easy to devise an optimal join plan based on the A* algorithm. Third, many optimization strategies can be designed on top of TwinTwigJoin, including order-aware cost reduction, workload skew reduction, and early filtering.

Note that most real-life data graphs are far from random. In this paper, we will first deliver the result of instance optimality of TwinTwigJoin by assuming that the data graph is a random graph. This not only provides the theoretic guarantee of the paradigm presented in the paper but also gives the foundation of our analysis of the power-law graphs. Later, we extend the results to the power-law graphs with the aim to cover many real applications, since many real-life large graphs are power-law graphs.

Contributions. We make the following contributions in this paper.

(1) A left-deep-join framework to join multiple edges in each round: In Section 3, we introduce a left-deep-join framework for subgraph enumeration in MapReduce, which generalizes the edge-based join to allow multiple edges (in the form of stars) to join in each round.

(2) A novel algorithm to ensure instance optimality: We propose a novel TwinTwigJoin algorithm in Section 5 following the left-deep-join framework, which uses TwinTwig as the right argument of the join in each MapReduce round. We analyze the cost of TwinTwigJoin based on the Erdös-Rényi random-graph model, upon which we prove that TwinTwigJoin is instance optimal in the left-deep-join framework under reasonable assumptions.We further develop an A*-based algorithm to compute the optimal join plan by defining a cost upper bound for any partial join. The algorithm can be adapted to any other graph model given that the cost upper bound for a partial join can be computed in the graph model.

(3) Three optimization strategies to further improve the algorithm: We explore three optimization strategies in Section 6, namely, order-aware cost reduction, workload skew reduction, and early filtering, to further improve the TwinTwigJoin algorithm. Order-aware cost reduction considers three types of TwinTwigs based on a predefined order of nodes in the data graph and pattern graph, which can be utilized to reduce the total computational cost. Workload skew reduction is used to reduce the workload skew caused by a few high-degree nodes in the data graph by partitioning their neighbors into multiple machines. Early filtering makes use of the free memory to further filter invalid partial results in early stages.

(4) Extension to power-law random graphs: We show in Section 7 how our algorithms and theoretical results can be adapted to the power-law graph model under a sound assumption.

(5) Extensive performance studies using web-scale real graphs: In Section 8, we conduct extensive performance studies in six real graphs with different graph properties, and the largest one of them contains billions of edges. The experimental results demonstrate that our TwinTwigJoin algorithm can achieve high scalability and outperforms all other state-of-the-art algorithms in all datasets.

2. PROBLEM DEFINITION

Subgraph Enumeration. We model a *data graph* as an undirected and unlabeled graph $G(V, E)$, where $V = V(G)$ represents the set of nodes and $E = E(G)$ represents the set of edges each of which connects two nodes in $V(G)$. We let $|V(G)| = N$ and $|E(G)| = M$, and assume $M > N$. We use $\{u_1, u_2, \dots, u_N\}$ to denote the set of nodes in G. For each $u_i \in V(G)$, we use $\mathcal{N}(u_i)$ to denote the set of neighbor nodes of u_i, and we use $d(u_i)$ to denote the degree of u_i, i.e., $d(u_i) = |\mathcal{N}(u_i)|$, and $d_{max} = \max_{u_i \in V(G)} d(u_i)$. We define $d = 2M/N$ to be the average degree of the data graph. A *subgraph* g of G is a graph such that $V(g) \subseteq V(G)$, $E(g) \subseteq E(G)$.

A *pattern graph* is an undirected, unlabeled and connected graph, denoted $P(V, E)$, where $V(P)$ represents the set of nodes and $E(P)$ represents the set of edges, and we let $|V(P)| = n$ and $|E(P)| = m$. We use $\{v_1, v_2, \dots, v_n\}$ to denote the set of nodes in P. For each $v_i \in V(P)$, $\mathcal{N}(v_i)$ and $d(v_i)$ are defined analogous to those defined in the data graph G. Note that it is trivial when P is a node or an edge, thus we assume $|V(P)| \geq 3$ in this paper.

Definition 2.1: (Match) Given a pattern graph P and a data graph G, a *match* f of P in G is a mapping from $V(P)$ to $V(G)$ such that the following two conditions hold:

- *(Conflict Freedom)* For any pair of nodes $v_i \in V(P)$ and $v_j \in V(P)$ $(i \neq j)$, $f(v_i) \neq f(v_j)$.
- *(Structure Preservation)* For any edge $(v_i, v_j) \in E(P)$, $(f(v_i), f(v_j)) \in E(G)$.

We use $f = (u_{k_1}, u_{k_2}, \dots, u_{k_n})$ to denote the match f, i.e., $f(v_i) = u_{k_i}$ for any $1 \leq i \leq n$. □

Definition 2.2: (Graph Isomorphism) Given two graphs g_i and g_j, g_i and g_j are *isomorphic*, if and only if there exists a match of g_i in g_j, and $|V(g_i)| = |V(g_j)|$ and $|E(g_i)| = |E(g_j)|$. □

Definition 2.3: (Subgraph Enumeration) Given a pattern graph P and a data graph G, *subgraph enumeration* is to enumerate all subgraphs g of G such that g is isomorphic to P. □

Definition 2.4: (Automorphism) Given a graph g, an automorphism of g is a match from g to itself. We use $\mathcal{A}(g)$ to denote the set of automorphisms for a graph g. □

Given a pattern graph P and a data graph G, if the total number of enumerated subgraphs is s then the total number of matches of P in G is $|\mathcal{A}(P)| \times s$. Since then, if P has only one automorphism, i.e., $|\mathcal{A}(P)| = 1$, the problem of subgraph enumeration is equivalent to enumerating all matches of P in G. In the following, for ease of analysis, we first assume that the pattern graph P has only one automorphism, i.e., $|\mathcal{A}(P)| = 1$, and thus we focus on enumerating all matches of P in G. In Section 5.4, we will discuss the general cases when $|\mathcal{A}(P)| \geq 1$.

Graph Storage. We assume the data graph G is stored in a distributed file system using adjacency lists, that is, for each node $u \in V(G)$, we store the adjacency list of u as a key-value pair $(u; \mathcal{N}(u))$ in the distributed file system.

Assumptions. In this paper, our theoretical results are derived based on the following assumptions:

- A_1: The data graph follows the Erdös Rényi random-graph model, which will be introduced in Section 5.2.
- A_2: The algorithm follows a left-deep-join framework, where the right join argument is a star. It will be further discussed in Section 3.
- A_3: The data graph is sparse; more specifically, the average degree $d = 2M/N < \sqrt{N}$.

Problem Statement. Given a data graph G stored in a distributed file system, and a pattern graph P, the purpose of this work is to enumerate all subgraphs of G that are isomorphic to P (based on Definiton 2.3) using MapReduce.

3. ALGORITHM FRAMEWORK

In this section, we introduce a left-deep-join-based framework for subgraph enumeration in MapReduce. Generally speaking, given a data graph G and a patten graph P, subgraph enumeration is pro-

Algorithm 1 SubgraphEnum(data graph G, pattern graph P)

```
1: compute a graph decomposition $\{p_0, p_1, \ldots, p_t\}$ of $P$;
2: for $i = 1$ to $t$ do
3:     $R(P_i) \leftarrow R(P_{i-1}) \bowtie R(p_i)$; (using map$^i$ and reduce$^i$)
4: return $R(P_t)$;
5: function map$^i$( key: $\emptyset$; value: either a match $f \in R(P_{i-1})$ when $i > 1$ or
   $(u, \mathcal{N}(u))$ for a node $u \in V(G)$ )
6: $\{v_{k_1}, v_{k_2}, \ldots, v_{k_l}\} \leftarrow V(P_{i-1}) \cap V(p_i)$;
7: if $i = 1$ then
8:     $G_u \leftarrow$ a graph formed by edges $(u, v)$ for $v \in \mathcal{N}(u)$;
9:     $R_u(P_0) \leftarrow$ all matches of $P_0$ in $G_u$;
10:    for all match $f \in R_u(P_0)$ do
11:        output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_l})); f)$;
12: if value is a match $f \in R(P_{i-1})$ then
13:    output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_l})); f)$;
14: else
15:    $G_u \leftarrow$ a graph formed by edges $(u, v)$ for $v \in \mathcal{N}(u)$;
16:    $R_u(p_i) \leftarrow$ all matches of $p_i$ in $G_u$;
17:    for all match $h \in R_u(p_i)$ do
18:        output $((h(v_{k_1}), h(v_{k_2}), \ldots, h(v_{k_l})); h)$;
19: function reduce$^i$( key: $r = (u_{k_1}, u_{k_2}, \ldots, u_{k_l})$; values: $F = \{f_1, f_2,$
    $\ldots\}, H = \{h_1, h_2, \ldots\}$ )
20: for all $(f, h) \in (F \times H)$ s.t. $(f - r) \cap (h - r) = \emptyset$ do
21:    output $(\emptyset; f \cup h)$;
```

cessed using a list of left-deep join operations, each of which is evaluated using one round of MapReduce. Before introducing the framework for subgraph enumeration, we first give the definitions of pattern decomposition, partial pattern, and partial result.

Definition 3.1: (Pattern Decomposition) Given a pattern graph P, a *pattern decomposition* of P, $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$ is a disjoint partition of the edges of P, such that p_i ($0 \le i \le t$) is a **star** (a tree of depth 1), and $V(p_i) \cap \bigcup_{0 \le j < i} V(p_j) \ne \emptyset$ ($i \ne 0$). □

Definition 3.2: (Partial Pattern P_i) Given a pattern decomposition $\{p_0, p_1, \ldots, p_t\}$ of P, a *partial pattern* P_i ($0 \le i \le t$) is a subgraph of P, such that $V(P_i) = \bigcup_{0 \le j \le i} V(p_j)$ and $E(P_i) = \bigcup_{0 \le j \le i} E(p_j)$. We have $P_0 = p_0$ and $P_t = P$. We use $\mathcal{D}_i = \{p_0, p_1, \ldots, p_i\}$ to denote a *partial pattern decomposition* of partial pattern P_i for any $0 \le i \le t$. □

According to the above definitions, we require that each decomposed unit p_i shares at least a common vertex with the partial pattern P_{i-1} for any $1 \le i \le t$.

Definition 3.3: (Partial Result $R(S)$) Given a subgraph S of the pattern graph P, and a data graph G, the *partial result* w.r.t. S, denoted as $R(S)$, is the set of matches of S in G. Obviously, $R(P)$ is the final result of the subgraph enumeration problem. □

The Framework. The framework of subgraph enumeration using MapReduce is shown in Algorithm 1. Given a graph G and a pattern P, we first compute a graph decomposition $\{p_0, p_1, \ldots, p_t\}$ of P which indicates a join plan (line 1). Then the algorithm is processed in t MapReduce rounds. Each round (lines 2-3) computes the partial result $R(P_i)$ by joining $R(P_{i-1})$ with $R(p_i)$, and obviously, $E(P_i) = E(P_{i-1}) \cup E(p_i)$ for $1 \le i \le t$. Each join operation is processed using MapReduce with map^i and reduce^i.

(Function map^i**)**: The function map^i is shown in lines 5-18 of Algorithm 1. The input of map^i is either a match $f \in R(P_{i-1})$ if $i > 1$, or $(u; \mathcal{N}(u))$ for a node $u \in V(G)$ (line 5). Both $R(P_{i-1})$ and G are stored in the distributed file system. We first calculate the join key $\{v_{k_1}, v_{k_2}, \ldots, v_{k_l}\}$ using $V(P_{i-1}) \cap V(p_i)$ (line 6). If $i = 1$, we need to compute the matches of P_0, $R_u(P_0)$, based on node u and its neighbors $\mathcal{N}(u)$, and output each such match (as a match in $R(P_0)$) along with the corresponding join key (lines 7-11). Then, if the input of map^i is a match $f \in R(P_{i-1})$, we simply output f along with the corresponding join key (lines 12-13). Otherwise, we compute the matches of p_i associated with u,

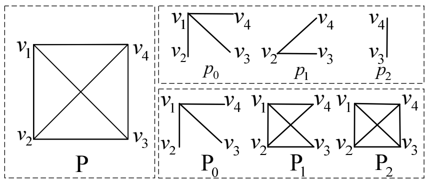


Figure 1: Pattern Decomposition and Query Processing

$R_u(p_i)$, as we do when we compute P_0 (lines 15-18).

(Function reduce^i**)**: The set of key-value pairs with the same key $r = (u_{k_1}, u_{k_2}, \ldots, u_{k_l})$ are processed using the same function reduce^i. There are two types of values, $F = \{f_1, f_2, \ldots\}$ and $H = \{h_1, h_2, \ldots\}$, generated by $R(P_{i-1})$ and $R(p_i)$ respectively. For each $(f, h) \in (F \times H)$ that shares the same join key, we output $f \cup h$ with the condition that $(f - r) \cap (h - r) = \emptyset$ to avoid node conflict (refer to the conflict freedom condition in Definiton 2.1)(lines 20-21).

Discussion. In the map^i phase of Algorithm 1, we need to compute $R(P_0)$ for $i = 1$ (line 9) and $R(p_i)$ for $1 \le i \le t$ (line 16) in G. Note that $R(P_0) = R(p_0)$, thus overall we need to compute $R(p_i)$ for $0 \le i \le t$ in G. We now discuss assumption A_2. Recall that G is stored as a set of key-value pairs $(u; \mathcal{N}(u))$ for $u \in V(G)$ in the distributed file system, and each key-value pair is processed by map^i separately according to the MapReduce framework. In this framework, *each p_i should be a **star***. As taken $(u; \mathcal{N}(u))$ as input, each map^i function can generate the matched stars rooted at u separately by enumerating the leaf nodes from $\mathcal{N}(u)$.

Example 3.1: In Fig. 1, we decompose the pattern graph into $\{p_0, p_1, p_2\}$. The corresponding partial patterns P_0, P_1, and P_2 are also presented. Based on the framework in Algorithm 1, the subgraph enumeration algorithm is processed in two MapReduce rounds. In the first round, we compute $R(P_1)$ using $R(P_0) \bowtie R(p_1)$ with $V(P_0) \cap V(p_1) = \{v_2, v_3, v_4\}$ as the join key. In the second round, we compute $R(P_2)$ using $R(P_1) \bowtie R(p_2)$ with $V(P_1) \cap V(p_2) = \{v_3, v_4\}$ as the join key. □

Remark. In Algorithm 1, we evaluate P using left-deep join based on the pattern decomposition $\mathcal{D}$. In addition to left-deep join, we can also use bushy join to process the tasks. Bushy join actually provides more varieties for an optimal join plan than its special case, left-deep join. However, we choose left-deep join in this paper due to the following aspects. First, as indicated in [32], left-deep join can still provide optimal solutions in many cases, especially when the join graph is highly connected. Second, left-deep join requires keeping much fewer partial results than bushy join. The partial results we need to keep are generated from only one iteration (the iteration prior to current one) in left-deep join but multiple iterations in bushy join. Finally and more importantly, it is much less expensive to compute an optimal join plan for left-deep join given its simpler computation structure.

4. EXISTING SOLUTIONS

In this section, we introduce three state-of-the-art algorithms for subgraph enumeration in MapReduce: EdgeJoin, StarJoin, and MultiwayJoin. Both EdgeJoin and StarJoin follow the left-deep-join framework (Algorithm 1) with different pattern-decomposition strategies. MultiwayJoin uses a new framework that enumerates all subgraphs in only one MapReduce round by duplicating edges of the data graph G.

Algorithm EdgeJoin. The EdgeJoin is proposed by Plantenga [29]. In EdgeJoin, each pattern graph P is decomposed into $\{p_0, p_1, \ldots, p_t\}$, where each p_i is an edge in $E(P)$. Thus, we have $t = m - 1$. The main drawback of EdgeJoin is that it may generate a large number of partial results since it cannot make full use of the structural information of the pattern graph, which can be explained via the following example.

Example 4.1: For a square pattern P where $V(p) = \{v_1, v_2, v_3, v_4\}$ and $E(P) = \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}$, the optimal pattern decomposition based on EdgeJoin is $p_0 = \{(v_1, v_2)\}, p_1 = \{(v_2, v_3)\}$, $p_2 = \{(v_3, v_4)\}$, $p_3 = \{(v_4, v_1)\}$. However, using this pattern-decomposition strategy, the algorithm executes in three MapReduce rounds, and the partial pattern P_3 is a path $\{(v_1, v_2), (v_2, v_3), (v_3, v_4)\}$ with length 3, which may result in a large number of partial results. A better strategy is to decompose P into two parts: $p_0 = \{(v_1, v_2), (v_2, v_3)\}$ and $p_1 = \{(v_3, v_4), (v_4, v_1)\}$, which can be processed in only one MapReduce round, and we can avoid processing P_3 as that in EdgeJoin. □

Algorithm StarJoin. The StarJoin algorithm decomposes the pattern graph into stars, where a star is a tree of depth 1. The star decomposition strategy is proposed by Sun et al. [33]. Given a pattern graph P and a node $v \in V(P)$, we denote $star(v)$ the star rooted at v with $\mathcal{N}(v)$ as its child nodes. According to [33], a star decomposition of P is defined as follows.

Definition 4.1: (Star Decomposition) Given a pattern graph P, a *star decomposition* is a decomposition $\{p_0, p_1, \ldots, p_t\}$ of P, such that there exists $\{v_{k_0}, v_{k_1}, \ldots, v_{k_t}\} \subseteq V(P)$ with $p_0 = star(v_{k_0})$, and $p_i = star(v_{k_i}) \setminus E(P_{i-1})$ for any $1 \leq i \leq t$. □

Compared to EdgeJoin, StarJoin can finish in fewer MapReduce rounds, however, StarJoin still suffers from the scalability problems due to the generation of many matches when evaluating a star with many edges.

Example 4.2: Fig. 1 shows an example of star decomposition for a 4-clique pattern graph P, in which p_0 is a star with three edges. In a social network such as Twitter, it is very common for a node to have more than 10,000 followers. As a result, this node with its followers will contribute to over 10^{12} matches of p_0. □

Algorithm MultiwayJoin. The MultiwayJoin algorithm is proposed by Afrati et al. [1]. MultiwayJoin enumerates subgraphs in the data graph using only one MapReduce round, while in order to do so, MultiwayJoin has to duplicate the edges several times in the map phase, and the number of duplications grows enormously with the size of the pattern graph. It is shown in [34] that MultiwayJoin can be efficient when P is a triangle. However, it will suffer from the scalability problem when P becomes more complex. For ease of analysis, we suppose P is a clique (complete graph) with n nodes. Let $b = \sqrt[n]{\#r}$, where $\#r$ is the number of reducers. With the optimal settings according to [1], the number of duplications for each edge of G is $\Theta(m \cdot b^{n-2}) = \Theta(n^2 \cdot b^{n-2})$, resulting in $\Theta(M \cdot n^2 \cdot b^{n-2})$ as a whole. Each reducer will hence receive $\Theta(\frac{M \cdot n^2 \cdot b^{n-2}}{\#r}) = \Theta(M \cdot \frac{n^2}{b^2})$ by average. There are two cases:

- *(Case-1: $b \leq n$)* A reducer will receive $\Theta(M \cdot \frac{n^2}{b^2}) \geq \Theta(M)$ edges, which is equivalent to holding the whole graph G.
- *(Case-2: $b > n$)* The total number of edge duplications is $\Theta(M \cdot n^2 \cdot b^{n-2}) > \Theta(M \cdot n^n)$, which is too large.

Obviously, both case-1 and case-2 are not scalable for either large data graph G or complex pattern graph P. Similar result can be derived when P is a general graph.

5. A NEW APPROACH

As discussed above, EdgeJoin, StarJoin, and MultiwayJoin will encounter scalability problems when the data graph is large or the pattern graph is complex. In this section, we propose a new algorithm TwinTwigJoin that follows the left-deep join framework introduced in Section 3 with a new pattern decomposition strategy, namely, TwinTwig decomposition. We first introduce the TwinTwig decomposition strategy, and analyze its optimality based on a variant of the random-graph model. Then we propose an optimal TwinTwig decomposition algorithm based on the A* framework. Finally, we discuss symmetry breaking to allow the pattern graph to have multiple automorphisms.

5.1 TwinTwig Decomposition

Definition 5.1: (TwinTwig Decomposition) A TwinTwig *decomposition* is a decomposition $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$ of pattern P such that each p_i ($0 \leq i \leq t$) is a TwinTwig, where a TwinTwig is either a single edge or two incident edges of a node. □

Our algorithm TwinTwigJoin is a left-deep-join algorithm (following Algorithm 1) based on TwinTwig decomposition. Obviously, TwinTwigJoin is a generalization of EdgeJoin. Compared to EdgeJoin, TwinTwigJoin can make use of more structural information of the pattern graph to reduce the size of the partial results. Compared to StarJoin, TwinTwigJoin can avoid joining a star with many edges by restricting the number of edges to be at most 2, and it is more flexible to select which one or two edge(s) of a star to join in a certain round to minimize the overall cost. Next, we introduce a special TwinTwig decomposition, namely, strong TwinTwig decomposition.

Definition 5.2: (Strong TwinTwig Decomposition) Let $\mathcal{D} = \{p_0, \ldots, p_t\}$ be a TwinTwig decomposition of P, a TwinTwig p_i ($1 \leq i \leq t$) is a *strong* TwinTwig if $|V(p_i) \cap V(P_{i-1})| \geq 2$, otherwise p_i is a *non-strong* TwinTwig. $\mathcal{D}$ is a *strong* TwinTwig *decomposition* if each p_i ($1 \leq i \leq t$) is a strong TwinTwig. The pattern P is *strong* TwinTwig *decomposable*, denoted SDEC, if there exists a strong TwinTwig decomposition of P. □

In the following, we will introduce the cost model and graph model, based on which we can prove the instance optimality of TwinTwigJoin under the assumptions introduced in Section 2.

5.2 Cost Analysis

Cost Model. Following the framework in Algorithm 1, for each MapReduce round i ($1 \leq i \leq t$), we consider three types of data, denoted $\mathcal{M}_i$, $\mathcal{S}_i$, and $\mathcal{R}_i$, which are defined as follows:

- $\mathcal{M}_i$ is the input of the i-th map phase. $\mathcal{M}_i$ includes all edges of graph G, and the partial result $R(P_{i-1})$ generated in the previous round (if $i > 1$). Thus, we have $|\mathcal{M}_1| = |E(G)|$ and $|\mathcal{M}_i| = |R(P_{i-1})| + |E(G)|$ for $i > 1$.
- $\mathcal{S}_i$ is the data transferred in the i-th shuffle phase, which is also the output of the i-th map phase as well as the input of the i-th reduce phase. $\mathcal{S}_i$ includes two parts, $R(P_{i-1})$ and $R(p_i)$, thus we have $|\mathcal{S}_i| = |R(P_{i-1})| + |R(p_i)|$.
- $\mathcal{R}_i$ is the output of the i-th reduce phase. $\mathcal{R}_i$ includes the set of partial matches $R(P_i)$, thus we have $|\mathcal{R}_i| = |R(P_i)|$.

There are many factors that can affect the efficiency of Algorithm 1, including I/O cost, communication cost, computational cost, number of MapReduce rounds, and workload balancing. We hence consider an overall cost $\mathcal{C}$ as follows:

$$\begin{aligned}\mathcal{C} = & \sum_{i=1}^{t}(|\mathcal{M}_i| + |\mathcal{S}_i| + |\mathcal{R}_i|) \\ = & \ 3\sum_{i=1}^{t}|R(P_i)| + |R(P_0)| + \sum_{i=1}^{t}|R(p_i)| + t|E(G)| - 2|R(P_t)| \\ = & \ 3\sum_{i=1}^{t}|R(P_i)| + \sum_{i=0}^{t}|R(p_i)| + t|E(G)| - 2|R(P_t)|\end{aligned}$$

Obviously, $\mathcal{C}$ is a comprehensive measurement of I/O cost, communication cost and computational cost, and it also implies the impact of the number of MapReduce rounds. Note that the last term

$2|R(P_t)| = 2|R(P)|$ is independent of the decomposition strategy, thus it can be removed from the cost function. Therefore, given any pattern decomposition $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$, the cost function, denoted as $\mathsf{cost}(\mathcal{D})$, can be defined as:

$$\mathsf{cost}(\mathcal{D}) = 3\sum_{i=1}^{t} |R(P_i)| + \sum_{i=0}^{t} |R(p_i)| + t|E(G)| \quad (1)$$

Similarly, for any $0 \le i \le t$, we can define the cost of a partial pattern decomposition $\mathcal{D}_i$ as:

$$\mathsf{cost}(\mathcal{D}_i) = 3\sum_{j=1}^{i} |R(P_j)| + \sum_{j=0}^{i} |R(p_j)| + i|E(G)| \quad (2)$$

For any $1 \le i \le t$, given that $\mathcal{D}_i = \mathcal{D}_{i-1} \cup \{p_i\}$, we have:

$$\mathsf{cost}(\mathcal{D}_i) = \mathsf{cost}(\mathcal{D}_{i-1}) + 3|R(P_i)| + |R(p_i)| + |E(G)| \quad (3)$$

Our aim is to find a decomposition $\mathcal{D}$ of the pattern graph P so that $\mathsf{cost}(\mathcal{D})$ is minimized.

Graph Model. In order to analyze the cost of different pattern-decomposition strategies, we will use two graph models to depict the data graph G, namely the Erdös-Rényi random-graph model [13], denoted ER model, and the power-law random-graph model [4], denoted PR model. In this paper, unless otherwise specified, we will use *random graph* to represent a graph constructed using the ER model, and *power-law random graph* for a graph constructed via PR model. As indicated by assumption A_1, we first focus on the case that the data graph is a random graph. Then we will extend our algorithm to handle the power-law random graphs in Section 7.

In the ER model, a graph is constructed by connecting nodes randomly. Each edge is included in the graph with probability ω independently from every other edges. Thus, for a data graph with N nodes and M edges, the probability ω can be calculated as: $\omega = \frac{2M}{N(N-1)}$, which can be approximated as $\frac{2M}{N^2}$ when N is large.

For simplicity, we use $|R(P)|$ to denote the expected number of matches for a pattern P in a random graph. The following lemma from [7] gives the value of $|R(P)|$.

Lemma 5.1: *Given a pattern graph P and a random graph G, if P is a connected graph, we have $|R(P)| = \frac{N!}{(N-n)!} \times \omega^m$.* □

Remark In practice, we often have $n \ll N$, hence we can evaluate $|R(P)|$ as:

$$|R(P)| = (2M)^m / N^{2m-n} \quad (4)$$

Results on SDEC Pattern Graph P. In order to show the instance optimality of the TwinTwig decomposition, we first study a special case, in which the pattern graph P is strong TwinTwig decomposable (SDEC). We have the following lemma.

Lemma 5.2: *Given an SDEC pattern graph P, suppose $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$ is a strong TwinTwig decomposition of P. For any partial pattern P_i $(1 \le i \le t)$, we have:*

$$|R(P_i)| \le |R(P_{i-1})| \times \tfrac{(2M)^2}{N^3} \le |R(p_0)| \times (\tfrac{(2M)^2}{N^3})^i. \quad \square$$

Proof: Suppose P_i contains n_i nodes and m_i edges, we have $|R(P_{i-1})| = \frac{(2M)^{m_{i-1}}}{N^{2m_{i-1}-n_{i-1}}}$ and $|R(P_i)| = \frac{(2M)^{m_i}}{N^{2m_i-n_i}}$. Let $\Delta m_i = m_i - m_{i-1}$ and $\Delta n_i = n_i - n_{i-1}$, we have:

$$|R(P_i)| = |R(P_{i-1})| \times (\frac{2M}{N^2})^{\Delta m_i} \times N^{\Delta n_i} \quad (5)$$

Since $\mathcal{D}$ is a strong TwinTwig decomposition, there are three cases for p_i $(1 \le i \le t)$:

- ($|E(p_i)| = 1$ and $|V(p_i) \cap V(P_{i-1})| = 2$): In this case, $\Delta m_i = 1$ and $\Delta n_i = 0$. It follows that:
$$|R(P_i))| = |R(P_{i-1})| \times \frac{2M}{N^2} < |R(P_{i-1})| \times \frac{(2M)^2}{N^3}.$$
- ($|E(p_i)| = 2$ and $|V(p_i) \cap V(P_{i-1})| = 2$): In this case, $\Delta m_i = 2$ and $\Delta n_i = 1$. It follows that:
$$|R(P_i))| = |R(P_{i-1})| \times (\frac{2M}{N^2})^2 \times N = |R(P_{i-1})| \times \frac{(2M)^2}{N^3}$$
- ($|E(p_i)| = 2$ and $|V(p_i) \cap V(P_{i-1})| = 3$): In this case, $\Delta m_i = 2$ and $\Delta n_i = 0$. It follows that:
$$|R(P_i))| = |R(P_{i-1})| \times (\frac{2M}{N^2})^2 < |R(P_{i-1})| \times \frac{(2M)^2}{N^3}$$

In all the above three cases, we have $|R(P_i)| \le |R(P_{i-1})| \times \frac{(2M)^2}{N^3}$. As a result, $|R(P_i)| \le |R(P_{i-1})| \times \frac{(2M)^2}{N^3} \le |R(P_{i-2})| \times (\frac{(2M)^2}{N^3})^2 \le \ldots \le |R(p_0)| \times (\frac{(2M)^2}{N^3})^i$. □

Corollary 5.1: *Under A_3, given an SDEC pattern graph P, suppose $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$ is a strong TwinTwig decomposition of P. For any partial pattern P_i $(1 \le i \le t)$, we have:*

$$|R(P_i)| \le |R(P_{i-1})| \le \ldots \le |R(P_0)| = |R(p_0)| \quad \square$$

Proof Sketch: By assumption A_3 ($d = 2M/N < \sqrt{N}$), we know that $\frac{(2M)^2}{N^3} = \frac{d^2}{N} < 1$. It is immediate that Corollary 5.1 holds according to Lemma 5.2. □

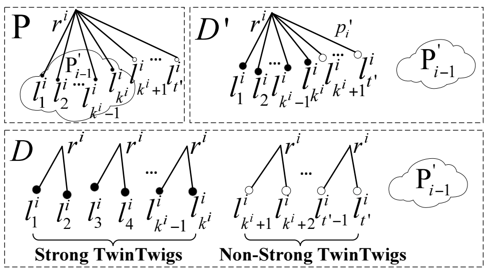


Figure 2: Constructing $\mathcal{D}$ based on $\mathcal{D}'$

The General Case. We prove the instance optimality of the general TwinTwig decomposition by showing that given any pattern decomposition $\mathcal{D}' = \{p'_0, p'_1, \ldots, p'_{t'}\}$, where each p'_i $(0 \le i \le t')$ is a star, we can construct a corresponding TwinTwig decomposition $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$ with $\mathsf{cost}(\mathcal{D}) \le \Theta(\mathsf{cost}(\mathcal{D}'))$.

We first introduce how to construct $\mathcal{D}$ based on $\mathcal{D}'$. For any $p'_i \in \mathcal{D}'$, let $\mathcal{D}^i = \{p^i_1, p^i_2, \ldots, p^i_{t_i}\}$ be a TwinTwig decomposition of p'_i which is constructed as follows: Suppose r^i is the root of p'_i and $\{l^i_1, l^i_2, \ldots, l^i_{t'_i}\}$ is the set of leaves of p'_i sorted by putting those nodes l^i_j with $l^i_j \in V(P'_{i-1})$ in the front (P'_{i-1} is the $i-1$-th partial pattern w.r.t. $\mathcal{D}'$), i.e., there exists a number k_i, s.t., if $1 \le j \le k_i$, $l^i_j \in V(P'_{i-1})$, and if $k_i < j \le t'_i$, $l^i_j \notin V(P'_{i-1})$. $\mathcal{D}^i = \{p^i_1, p^i_2, \ldots, p^i_{t_i}\}$ is constructed as follows:

- If t'_i is an even number, then $t_i = \frac{t'_i}{2}$, and p^i_j $(1 \le j \le t_i)$ is a TwinTwig with root r^i and two leaves l^i_{2j-1} and l^i_{2j}.
- If t'_i is an odd number, then $t_i = \frac{t'_i+1}{2}$, and p^i_j $(1 \le j \le t_i - 1)$ is a TwinTwig with root r^i and two leaves l^i_{2j-1} and l^i_{2j}, and $p^i_{t_i}$ is a TwinTwig with only one edge $(r^i, l^i_{t'_i})$.

In other words, $\mathcal{D}^i$ is constructed by generating strong TwinTwigs followed by non-strong TwinTwigs. After constructing $\mathcal{D}^i$ for all $0 \le i \le t'$, we have $\mathcal{D}$ by combining all $\mathcal{D}^i$, i.e., $\mathcal{D} = \bigcup_{i=0}^{t'} \mathcal{D}^i$. The construction of $\mathcal{D}$ from $\mathcal{D}'$ is illustrated in Fig. 2.

We show the instance optimality of a general TwinTwig decomposition using the following theorem:

Theorem 5.1: *Under the assumption A_3, given a pattern decomposition $\mathcal{D}' = \{p'_0, p'_1, \ldots, p'_{t'}\}$ where each p'_i $(0 \le i \le t')$ is a star, let $\mathcal{D}$ be the TwinTwig decomposition constructed based on $\mathcal{D}'$ using the above method, then $\mathsf{cost}(\mathcal{D}) \le \Theta(\mathsf{cost}(\mathcal{D}'))$.* □

Proof: For any pattern decomposition $\mathcal{D}$, we divide $\mathsf{cost}(\mathcal{D}) = 3\Sigma_{i=1}^{t}|R(P_i)| + \Sigma_{i=0}^{t}|R(p_i)| + t|E(G)|$ (Eq. 1) into two parts:

- $\mathsf{cost}_1(\mathcal{D}) = \Sigma_{i=0}^{t}\, |R(p_i)| + t|E(G)|$.
- $\mathsf{cost}_2(\mathcal{D}) = 3\,\Sigma_{i=1}^{t}\, |R(P_i)|$.

Accordingly, we divide the proof into two parts:

(Part 1): We prove $\mathsf{cost}_1(\mathcal{D}) \le \Theta(\mathsf{cost}_1(\mathcal{D}'))$. We only need to prove $\mathsf{cost}_1(\mathcal{D}^i) \le \Theta(\mathsf{cost}_1(\{p'_i\}))$ for each $0 \le i \le t'$. Note that when $|E(p'_i)| \le 2$, $\mathsf{cost}_1(\mathcal{D}^i) = \mathsf{cost}_1(\{p'_i\})$, thus, we only consider $|E(p'_i)| \ge 3$. In this case, we have:

- $\mathsf{cost}_1(\mathcal{D}^i) \le \Theta(t'_i \cdot d^2 \cdot N)$. According to Eq. 4, we know that each pattern $p^i_j \in D^i$ is a TwinTwig with $|R(p^i_j)| \le \frac{(2M)^2}{N} = \Theta(d^2 \cdot N)$. Hence, we have:

$$cost_1(D^i) = \sum_{j=1}^{\lceil t'_i/2 \rceil} (|R(p^i_j)| + |E(G)|) \le \Theta(t'_i \cdot d^2 \cdot N)$$

- $\mathsf{cost}_1(\{p'_i\}) \ge \Theta(t'_i \cdot d^3 \cdot N)$. This is because:

$$\begin{aligned}\mathsf{cost}_1(\{p'_i\}) &\ge |R(p'_i)| = d^{t'_i} \times N \ge (t'_i - 2) \times d^3 \times N \\ &\ge t'_i/3 \times d^3 \times N \quad (\text{by } t'_i = |E(p'_i)| \ge 3) \\ &= \Theta(t'_i \cdot d^3 \cdot N)\end{aligned}$$

Thus, $\mathsf{cost}_1(\mathcal{D}^i) \le \Theta(\mathsf{cost}_1(\{p'_i\}))$.

(Part 2): We prove $\mathsf{cost}_2(\mathcal{D}) = \Theta(\mathsf{cost}_2(\mathcal{D}'))$. We reformulate $\mathsf{cost}_2(\mathcal{D}')$ as $3(\frac{p'_0}{2} + \frac{\Sigma_{i=1}^{t'}|R(P'_{i-1})|+|R(P'_i)|}{2} + \frac{|R(P'_{t'})|}{2})$. Thus:

$$\mathsf{cost}_2(\mathcal{D}') = \Theta(\sum_{i=1}^{t'}(|R(P'_{i-1})| + |R(P'_i)|)) \tag{6}$$

Note that in $\mathcal{D}$ that is constructed based on $\mathcal{D}'$, we will gradually combine p^i_1, p^i_2, $\dots$, $p^i_{t_i}$ to P'_{i-1} in order to get P'_i. Hence, the term $|R(P'_{i-1})| + |R(P'_i)|$ for each $1 \le i \le t'$ in $\mathsf{cost}_2(\mathcal{D}')$ is replaced by:

$$\begin{aligned}\mathsf{cost}^i_2(\mathcal{D}) = |R(P'_{i-1})| + |R(P'_{i-1} \cup p^i_1)| \\ + \dots + |R(P'_{i-1} \cup p^i_1 \cup \dots \cup p^i_{t_i-1})| + |R(P'_i)|\end{aligned} \tag{7}$$

Recall that there exists a k_i such that, when $1 \le j \le k_i$, p^i_j is a strong TwinTwig, and when $k_i < j \le t_i$, p^i_j is a non-strong TwinTwig. Let $x = k_i$ and $y = t_i - k_i$, then there are $x + y + 1$ terms in $\mathsf{cost}^i_2(\mathcal{D})$. We have,

- (S_1): The sum of the first $x+1$ terms in $\mathsf{cost}^i_2(\mathcal{D})$ is $\Theta(|R(P'_{i-1})|)$. Since each p^i_j is a strong TwinTwig, according to Lemma 5.2 and Corollary 5.1, when j increases, the size of the j-th term decreases exponentially with a rate $\le \frac{(2M)^2}{N^3} < 1$, thus, statement S_1 holds.
- (S_2): The sum of the last y terms in $\mathsf{cost}^i_2(\mathcal{D})$ is $\Theta(|R(P'_i)|)$. Since each p^i_j is a non-strong TwinTwig, according to Eq. 5, when j increases, the size of the j-th term increases exponentially with a rate $\ge d > 1$, thus, statement S_2 holds.

Based on S_1 and S_2, we have $\mathsf{cost}_2(\mathcal{D}) = \Theta(\mathsf{cost}_2(\mathcal{D}'))$, and according to Part 1 and Part 2, Theorem 5.1 holds. □

5.3 Optimal Decomposition by A*

In this subsection, we will show how to construct an optimal TwinTwig decomposition for any pattern graph P using an A*-based algorithm.

The Cost Function. The key of the A*-based algorithm is to find a cost function for each partial solution, which defines the priority of the partial solution to be expanded to form the final solution. In the subgraph enumeration problem, for any partial TwinTwig decomposition $\mathcal{D}_i$ of P (refer to Definiton 3.2), we need to define a cost function $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$, which is the cost lower bound for any TwinTwig decomposition of P expanded from $\mathcal{D}_i$. We compute $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ using dynamic programming. Given a partial pattern P_i, we use $\underline{\Delta\mathsf{cost}}(P_i, \Delta m, \Delta n)$ to denote the lower bound of the increased cost when adding any Δm edges and Δn nodes into the partial pattern P_i. Let $\mathsf{card}(m, n) = |R(P)|$ be the number of

Algorithm 2 Optimal-Decomp(data graph G, pattern graph P)

```
1: $\mathcal{H} \leftarrow \emptyset$;
2: for all TwinTwig $p$ in $P$ do
3:     $\mathcal{H}.push((p, \{p\}, \underline{\mathsf{cost}}(\{p\}, P)))$;
4: $(P', \mathcal{D}', \underline{\mathsf{cost}}(\mathcal{D}', P)) \leftarrow \mathcal{H}.pop()$;
5: while $P' \neq P$ do
6:     for all TwinTwig $p$ with $V(p) \cap V(P') \neq \emptyset$ and $E(p) \cap E(P') = \emptyset$ do
7:         if $\mathcal{H}.find(P' \cup p) \neq \emptyset$ then
8:             $\mathcal{H}.update(P' \cup p, \mathcal{D}' \cup \{p\}, \underline{\mathsf{cost}}(\mathcal{D}' \cup \{p\}, P))$;
9:         else $\mathcal{H}.push((P' \cup p, \mathcal{D}' \cup \{p\}, \underline{\mathsf{cost}}(\mathcal{D}' \cup \{p\}, P)))$;
10:    $(P', \mathcal{D}', \underline{\mathsf{cost}}(\mathcal{D}', P)) \leftarrow \mathcal{H}.pop()$;
11: return $\mathcal{D}'$;
```

matches of any connected pattern graph P with m edges and n nodes, according to Eq. 4, we have:

$$\mathsf{card}(m, n) = (2M)^m / N^{2m-n} \tag{8}$$

In the dynamic algorithm, the initial state is $\underline{\Delta\mathsf{cost}}(P_i, 0, 0) = 0$, and according to Eq. 3, the transaction function is formulated as:

$$\begin{aligned}\underline{\Delta\mathsf{cost}}(P_i, \Delta m, \Delta n) = \min\{\underline{\Delta\mathsf{cost}}(P_i, \Delta m - a, \Delta n - b) \\ + 3 \times \mathsf{card}(|E(P_i)| + \Delta m, |V(P_i)| + \Delta n) + \mathsf{card}(a, b) \\ + M \mid \forall 1 \le a \le 2, 0 \le b \le a, a \le \Delta m, b \le \Delta n\}\end{aligned}$$

The conditions $1 \le a \le 2$ and $0 \le b \le a$ are required to guarantee that we join a TwinTwig each time. Accordingly, $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ can be calculated as:

$$\begin{aligned}\underline{\mathsf{cost}}(\mathcal{D}_i, P) = \mathsf{cost}(\mathcal{D}_i) \\ + \underline{\Delta\mathsf{cost}}(P_i, |E(P)| - |E(P_i)|, |V(P)| - |V(P_i|)\end{aligned} \tag{9}$$

Note that $\underline{\Delta\mathsf{cost}}(P_i, \Delta m, \Delta n)$ is only dependent on $|E(P_i)|$ and $|V(P_i)|$, thus we can denote $\underline{\Delta\mathsf{cost}}(\ P_i, \Delta m, \Delta n)$ of any P_i as:

$$\underline{\Delta\mathsf{cost}}(m', n', \Delta m, \Delta n)$$

where $m' = |E(P_i)|$ and $n' = |V(P_i)|$. As a result, given a data graph G, we can precompute $\underline{\Delta\mathsf{cost}}(m', n', \Delta m, \Delta n)$ for all possible m', n', Δm, and Δn, given that $\underline{\Delta\mathsf{cost}}(m', n', \Delta m, \Delta n)$ is query independent. The time complexity and space complexity for the precomputation are both $O((\overline{m} \cdot \overline{n})^2)$, where $\overline{m}$ and $\overline{n}$ are the upper bounds on m' and n' respectively. In such a way, given any $\mathcal{D}_i$ and P, suppose $\mathsf{cost}(\mathcal{D}_i)$ is computed, then $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ can be computed in $O(1)$ time.

The Algorithm. The A* algorithm to compute the optimal decomposition is shown in Algorithm 2. Let $\mathcal{H}$ be a heap in which each entry has the form $(P', \mathcal{D}', \underline{\mathsf{cost}}(\mathcal{D}', P))$, where P' is a partial pattern and $\mathcal{D}'$ is the corresponding partial TwinTwig decomposition. The top entry in $\mathcal{H}$ is a pattern decomposition $\mathcal{D}'$ with the minimum $\underline{\mathsf{cost}}(\mathcal{D}', P)$. The algorithm follows a typical A* framework that (1) iteratively pops the minimum entry (line 4 and line 10), (2) expands the entry with one TwinTwig (line 6), and (3) updates the new entry if the corresponding partial pattern is already in $\mathcal{H}$ and current cost is smaller than the existing one (line 7-8), or (4) pushes the new entry into $\mathcal{H}$ if the corresponding partial pattern is not in $\mathcal{H}$ (line 9). The algorithm stops when the popped partial pattern is the pattern graph P (line 5) and returns the last popped $\mathcal{D}'$ as the optimal TwinTwig decomposition (line 11).

Lemma 5.3: *The space complexity and time complexity of Algorithm 2 are $O(2^m)$ and $O(\overline{d} \cdot m \cdot 2^m)$ respectively, where $\overline{d} = \max_{v \in V(P)} d(v)$.* □

Proof Sketch: We first prove the space complexity. Each entry $(P', \mathcal{D}', \underline{\mathsf{cost}}(\mathcal{D}', P))$ in $\mathcal{H}$ is uniquely identified by the partial pattern P', and there are at most 2^m partial patterns, which consumes at most $O(2^m)$ space. Note that each P' and $\mathcal{D}'$ can be stored using constant space by only keeping the last TwinTwig p that generates P' and $\mathcal{D}'$, and a link to the entry identified by $P' - p$.

Next we prove the time complexity. Let s be the possible number of TwinTwigs in P, we have:

$$s = \Sigma_{v \in V(P)} d(v)^2 \le \Sigma_{v \in V(P)} d(v) \times \overline{d} = 2m \times \overline{d}.$$

When an entry is popped out from $\mathcal{H}$, it can be expanded at most s times. Using a Fibonacci heap, *pop* works in $\log(|\mathcal{H}|)$ time, and *update* and *push* both work in $O(1)$ time. Thus the overall time complexity is:

$$O(2^m \cdot (s+\log(|\mathcal{H}|))) = O(2^m \cdot (s+\log(2^m))) = O(\overline{d} \cdot m \cdot 2^m) \square$$

Discussion. In practice, the processing time for Algorithm 2 is much smaller than $O(\overline{d} \cdot m \cdot 2^m)$ since $\mathcal{H}$ only keeps connected subgraphs of P that can potentially result in the optimal solution.

5.4 Symmetry Breaking

Graph automorphism is also known as graph symmetry [12, 16]. In this subsection, we show how to use symmetry-breaking to remove the assumption that the pattern graph P has no non-trivial automorphism. When $|\mathcal{A}(P)| > 1$, by directly applying Algorithm 1, each enumerated subgraph will be duplicated for $|\mathcal{A}(P)|$ times. The primary goal is to effectively prevent duplicates (i.e., a subgraph of a data graph will not be enumerated twice) while not missing results. For this purpose, we implemented the techniques in [16] that ensures 1) no duplicates are generated while enumerating, and 2) no missing results. Below we provide a brief description.

We assume that there is a total order (defined by $\prec$) among all nodes in the data graph G. The symmetry-breaking paradigm is performed by assigning a partial order (defined by $<$) among some pairs of nodes in the pattern graph P. The algorithm to compute the partial order for symmetry-breaking has been introduced in details in [16].

Given a partial order in P, a *match* is redefined from Definiton 2.1 by adding a new *order preservation* constraint, that is, for any pair of nodes $v_i \in V(P)$ and $v_j \in V(P)$, if $v_i < v_j$, then $f(v_i) \prec f(v_j)$.

Algorithm 1 can be extended to handle the partial order as follows: In the map^i phase, when computing $R(p_i)$ (line 9, line 16), we make sure that each match satisfies the *order preservation* constraint. In the reduce^i phase, in line 21, we only output those $f \cup h$ that satisfy the *order preservation* constraint. In Section 6.1, we will discuss how to use the partial order to further optimize pattern decomposition.

6. OPTIMIZATION STRATEGIES

In this section, we discuss three optimization strategies to further improve our subgraph enumeration algorithm, namely, order-aware cost reduction, workload skew reduction, and early filtering.

6.1 Order-aware Cost Reduction

In this subsection, we discuss how to make use of the partial order to further reduce the computational cost. We first consider a motivating example: Let the pattern graph P be a triangle of three nodes v_1, v_2, and v_3, with $v_1 < v_2 < v_3$ for symmetry-breaking. By TwinTwig decomposition, P is decomposed into $\mathcal{D} = \{p, e\}$, where p is a two-edge TwinTwig, and e is a single edge. According to Eq. 1, we can derive $\mathsf{cost}(\mathcal{D}) = 3|R(P)| + |R(p)| + 2M$. Since $|R(P)|$ and M are fixed, $\mathsf{cost}(\mathcal{D})$ is only dependent on p which has 3 choices: $p_1 = \{(v_1, v_2), (v_1, v_3)\}$, $p_2 = \{(v_1, v_2), (v_2, v_3)\}$, and $p_3 = \{(v_1, v_3), (v_2, v_3)\}$. Let the data graph G be a star with a root node r and $N-1$ leaf nodes. Obviously, in such a case $|R(P)| = 0$. Consider the following 3 cases C_1, C_2 and C_3:

- C_1: r has the largest order in $V(G)$. In this case, $|R(p_1)| = |R(p_2)| = 0$ and $|R(p_3)| = \Theta(N^2)$.
- C_2: r has the smallest order in $V(G)$. In this case, $|R(p_1)| = \Theta(N^2)$ and $|R(p_2)| = |R(p_3)| = 0$.
- C_3: r has the median order in $V(G)$. In this case, $|R(p_1)| = |R(p_2)| = |R(p_3)| = \Theta(N^2)$.

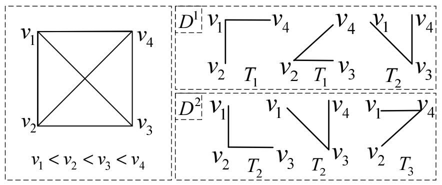


Figure 3: Order-Aware Decomposition

In both C_1 and C_2, we can find a p with $|R(p)| = 0$ which is optimal. This extreme example motivates us to link the order of nodes in $V(G)$ to their degrees. Specifically, we assign a new total order of nodes in $V(G)$ by redefining the operator $\prec$ as follows:

Definition 6.1: (Operator $\prec$) For any two nodes u_i and u_j in $V(G)$, $u_i \prec u_j$ if and only if one of the two conditions holds:

- $d(u_i) < d(u_j)$,
- $d(u_i) = d(u_j)$ and $id(u_i) < id(u_j)$.

Where $id(u)$ is the unique identity of node u ($\in V(G)$). Obviously, the operator $\prec$ specifies a total order for nodes in $V(G)$. $\square$

Given the new total order for $V(G)$, for any $u \in V(G)$, we let $\mathcal{N}^+(u) = \{u' \mid u' \in \mathcal{N}(u),\ u \prec u'\}$ and $\mathcal{N}^-(u) = \{u' \mid u' \in \mathcal{N}(u),\ u' \prec u\}$. We then define $d^+(u) = |\mathcal{N}^+(u)|$ and $d^-(u) = |\mathcal{N}^-(u)|$, and $d^+_{max} = \max_{u \in V(G)} d^+(u)$ and $d^-_{max} = \max_{u \in V(G)} d^-(u)$. For a two-edge TwinTwig $p = \{(v, v_1), (v, v_2)\}$, we consider the following three types of orders:

- T_1: $v < v_1 < v_2$ or $v < v_2 < v_1$;
- T_2: $v_1 < v < v_2$ or $v_2 < v < v_1$;
- T_3: $v_1 < v_2 < v$ or $v_2 < v_1 < v$.

Let p^{T_1}, p^{T_2}, and p^{T_3} be TwinTwigs of types T_1, T_2, and T_3 respectively. We have the following results:

- $|R(p^{T_1})| = O(\Sigma_{u \in V(G)} (d^+(u))^2) = O(\alpha \cdot M)$;
- $|R(p^{T_2})| = O(\Sigma_{u \in V(G)} (d^+(u) \cdot d^-(u))) = O(d^+_{max} \cdot M)$;
- $|R(p^{T_3})| = O(\Sigma_{u \in V(G)} (d^-(u))^2) = O(d^-_{max} \cdot M)$.

Where α is the *arboricity* of the graph G and $\alpha \leq d^+_{max} \leq d^-_{max}$ according to [8]. Thus, when selecting TwinTwigs for joining, p^{T_1} is preferable to p^{T_2}, followed by p^{T_3}. We give an example below to show the three types of TwinTwig$s$.

Example 6.1: Fig. 3 shows a 4-clique pattern graph P with order $v_1 < v_2 < v_3 < v_4$, and two decomposition plans $\mathcal{D}^1$ and $\mathcal{D}^2$, both of which are strong TwinTwig decompositions. However, $\mathcal{D}^1$ contains two p^{T_1}s and one p^{T_2}, and $\mathcal{D}^2$ contains two p^{T_2}s and one p^{T_3}. Obviously, $\mathcal{D}^1$ is better than $\mathcal{D}^2$. $\square$

Order-aware TwinTwig Decomposition. We discuss how to modify Algorithm 2 for TwinTwig decomposition by taking the partial order into consideration. Recall that Algorithm 2 only depends on the cost function $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ (Eq. 9) for any partial TwinTwig decomposition $\mathcal{D}_i$, and $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ is calculated based on $\mathsf{cost}(\mathcal{D}_i)$ and $\underline{\Delta\mathsf{cost}}(P_i, \Delta m, \Delta n)$, both of which are originated from Eq. 1. Thus, we only need to reestimate $|R(p_i)|$ and $|R(P_i)|$ for any p_i and partial pattern P_i by taking the partial order into consideration.

(Reestimate $|R(p_i)|$): Let $p_i = \{(v, v_1), (v, v_2)\}$. In order to calculate $|R(p_i)|$, we precompute $|R(p^{T_1})|$, $|R(p^{T_2})|$, and $|R(p^{T_3})|$. If p_i only contains 1 edge, then $|R(p_i)| = M$; otherwise, $|R(p_i)|$ can be calculated from $|R(p^{T_1})|$, $|R(p^{T_2})|$, and $|R(p^{T_3})|$ depending on the partial orders defined on $V(p_i)$. For instance, if the partial order is only defined on one pair $v < v_1$ in p_i, then $|R(p_i)|$ can be calculated as $2 \times |R(p^{T_1})| + |R(p^{T_2})|$.

(Reestimate $|R(P_i)|$): $|R(P_i)|$ is hard to calculate when the partial order is involved, however, after each round of join, we try to make use of the updated information to better estimate $|R(P_i)|$ at runtime. Specifically, after the j-th round of join, suppose the current partial pattern is P_j, and $|R(P_j)|$ has been accurately calculated. Then for any possible future partial pattern P_i which is a supergraph of P_j, according to Eq. 5, $|R(P_i)|$ can be calculated as:

$$|R(P_i)| = |R(P_j)| \times (\frac{2M}{N^2})^{|E(P_i)|-|E(P_j)|} \times N^{|V(P_i)|-|V(P_j)|} \quad (10)$$

Based on the reestimating technique, Algorithm 1 is modified as follows: In the first round, it computes the optimal decomposition plan using the A* algorithm (Algorithm 2) directly, and then processes the first MapReduce round accordingly. In the following round i ($i > 1$), before processing MapReduce, the algorithm recomputes the optimal decomposition using the A* algorithm with the reestimating technique where each $|R(P_j)|$ for $0 \le j < i$ is replaced by the accurate value. In this way, the partial order is involved in Algorithm 1.

6.2 Workload Skew Reduction

For many real graphs, it is very common that a small number of nodes in a graph have very high degrees. Given a data graph G, we denote the high-degree nodes by V^H (e.g., nodes with degree larger than $\sqrt{M}$). Recall that G is stored in a distributed file system using adjacency lists in the form $(u; \mathcal{N}(u))$ for each $u \in V(G)$. For a two-edge TwinTwig p, evaluating p on the adjacency list $(u; \mathcal{N}(u))$ will generate $\Theta(d(u)^2)$ matches, rendering very high workloads in the machines that are processing high-degree nodes. This motivates us to consider the workload balancing issue. In the following, we discuss our strategy to reduce the workload skew.

Suppose there are λ machines in the system, for any $u \in V^H$, instead of using $(u, \mathcal{N}(u))$, we divide $\mathcal{N}(u)$ uniformly into β partitions: $\mathcal{N}(u) = \{\mathcal{N}_1(u), \mathcal{N}_2(u), \dots, \mathcal{N}_\beta(u)\}$. Note that we cannot simply distribute the β partitions into the λ machines. Because if so, given a TwinTwig $p = \{(v, v_1), (v, v_2)\}$, the match $f = (u, u_1, u_2) \in R(p)$ with $u_1 \in \mathcal{N}_i(u)$ and $u_2 \in \mathcal{N}_j(u)$ ($i \ne j$) cannot be generated by any machine. To handle this, we create $\frac{\beta \times (\beta+1)}{2}$ partitions in the following two sets $S_1(u)$ and $S_2(u)$, and distribute the partitions uniformly into the λ machines.

- $S_1(u) = \{(u; \mathcal{N}_i(u)) | 1 \le i \le \beta\}$;
- $S_2(u) = \{(u; (\mathcal{N}_i(u), \mathcal{N}_j(u))) | 1 \le i < j \le \beta\}$.

With $S_1(u)$ and $S_2(u)$, when evaluating a TwinTwig with one edge, only $S_1(u)$ needs to be used; and when evaluating a TwinTwig with two edges, both $S_1(u)$ and $S_2(u)$ need to be used. By setting $\beta = \Theta(\sqrt{\lambda})$, the number of partitions becomes $\Theta(\lambda)$. As a result, each machine just keeps a constant number of partitions in $S_1(u) \cup S_2(u)$ uniformly. It is easy to verify that the total space used to keep $S_1(u)$ and $S_2(u)$ is $\Theta(\sqrt{\lambda} \cdot |\mathcal{N}(u)|)$.

6.3 Early Filtering

Recall that Algorithm 1 only requires very small memory in both map^i and reduce^i. This motivates us to make use of the remaining memory for further optimization. Specifically, we use bloom filter [6] to prune the invalid partial matches in early stages of the algorithm to reduce the cost. Generally speaking, given a set S and a memory budget $\mathcal{M}$, a bloom filter for S denoted as $\mathcal{G}(S)$, can be created using no more than $\mathcal{M}$ memory such that given any element e, it can answer whether $e \in S$ with no false negatives and a small probability of false positives denoted as fp . There is a trade-off between the size of the memory $\mathcal{M}$ and the probability of false positives fp.

In our approach, we create a bloom filter $\mathcal{G}(E(G))$ in every machine of the system, and we use the bloom filter $\mathcal{G}(E(G))$ for the following two types of early filtering mechanisms in Algorithm 1:

- *(Map Side Filtering)*: When evaluating $R(p_i)$ for any TwinTwig $p_i = \{(v, v_1), (v, v_2)\}$ in the map phase, if $(v_1, v_2) \in E(P)$, then any match (u, u_1, u_2) with $(u_1, u_2) \notin E(G)$ is pruned by $\mathcal{G}(E(G))$ with probability $1 - fp$.
- *(Reduce Side Filtering)*: When evaluating $R(P_i)$ for any partial pattern P_i in the reduce phase, for any $(v_1, v_2) \in E(P) - E(P_i)$ with $v_1 \in V(P_i)$ and $v_2 \in V(P_i)$, any partial match $f \in R(P_i)$ with $(f(v_1), f(v_2)) \notin E(G)$ is pruned by $\mathcal{G}(E(G))$ with probability $1 - fp$.

Obviously, early filtering does not affect the correctness of Algorithm 1 since only invalid partial patterns are pruned by the bloom filter $\mathcal{G}(E(G))$. Note that early filtering can be applied for all the three algorithms EdgeJoin, StarJoin, and TwinTwigJoin.

Example 6.2: Suppose the pattern graph P is a triangle of three nodes. We can decompose P into $\mathcal{D} = \{p, e\}$ where p is a two-edge TwinTwig and e is a single edge. According to Eq. 1, we have $\mathsf{cost}(\mathcal{D}) = 3|R(P)| + |R(p)| + 2M$. Without early filtering, it is possible that $|R(p)|$ dominates the whole cost with $|R(p)| >> |R(P)|$ and $|R(p)| >> M$. Suppose we use $\mathcal{G}(E(G))$ with $fp = 0.1$, then $R(p)$ is filtered in the map phase with only 0.1 ratio of false positives, i.e., $|R(p)| = 1.1|R(P)|$, as a result $|\mathsf{cost}(\mathcal{D})| = \Theta(|R(P)| + M)$, which is optimal since M is the size of the input and $|R(P)|$ is the size of the final output. □

7. HANDLING POWER-LAW GRAPHS

We model the data graph G of N nodes and M edges as a power-law random graph according to [4]. We consider a non-increasing degree sequence $\{w_1, w_2, \dots, w_N\}$ that satisfies the power-law distribution, that is, the number of nodes with a certain degree x is proportional to $x^{-\beta}$, where β is the power-law exponent. For any pair of nodes u_i and u_j in a power-law random graph, the edge between u_i and u_j is independently assigned with probability $P_{i,j} = w_i w_j \rho$, where $\rho = 1/\Sigma_{i=1}^N w_i = 1/2M$. It is easy to verify that the expected degree of u_i is equal to w_i for any $1 \le i \le N$. We define the average degree as $d = (\Sigma_{i=1}^N w_i)/N$, and the maximum degree as d_{max}. Note that we only consider $2 < \beta < 3$ in this paper, as many real graphs have the power-law exponent in this range [9, 10]. We engage the small-degree assumption A_4 in this model as follows:

$$A_4 : d_{max} \le \sqrt{N}$$

Though this assumption may not be satisfied in some real graphs, in the experiment, we will show the intermediate results from the vertices with degree $\le \sqrt{N}$ play a dominant role in the total intermediate results.

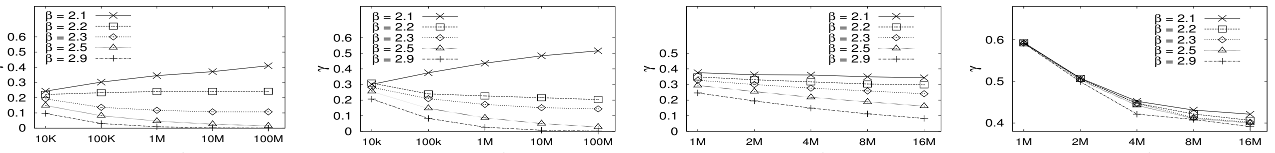
Instance Optimality. In order to show the instance optimality, we will prove that Theorem 5.1 holds in a power-law random graph under the small-degree assumption A_4, following the same proof structure as that in the proof of Theorem 5.1. Similarly, we divide the proof into the following two parts: In part 1, we prove that $\mathsf{cost}_1(\mathcal{D}) \le \Theta(\mathsf{cost}_1(\mathcal{D}'))$, and in part 2, we prove that $\mathsf{cost}_2(\mathcal{D}) = \Theta(\mathsf{cost}_2(\mathcal{D}'))$. In order to prove part 2, we still compare Eq. 6 and Eq. 7, and then prove the two cases, namely, S_1: the size of the results decreases after joining a strong TwinTwig; S_2: the size of the results increases after joining a non-strong TwinTwig. The detailed proof is as follows.

(Part 1): Let p be a two-edge TwinTwig, we have:

$$\begin{aligned} \mathsf{cost}_1(\mathcal{D}^i) &= \Theta(|R(p)| \cdot t'_i) \text{ and,} \\ \mathsf{cost}_1(\{p'_i\}) &= \Theta(|R(p)| \cdot \mathbb{E}[d(u)^{t'_i-2}]) \\ &\ge \Theta(|R(p)| \cdot \mathbb{E}[d(u)]^{t'_i-2}) = \Theta(|R(p)| \cdot d^{t'_i-2}) \end{aligned}$$

where $\mathbb{E}[d(u)]$ is the expected degree for an arbitrary node u in $V(G)$. Given that $d \ge 2$ and $t'_i \ge 3$, it is easy to see that $\mathsf{cost}_1(\mathcal{D}^i) \le \mathsf{cost}_1(\{p'_i\})$ for each $0 \le i \le t'$, which results in $\mathsf{cost}_1(\mathcal{D}) \le \Theta(\mathsf{cost}_1(\mathcal{D}'))$. Therefore, part 1 is proved.

(Part 2): For a certain pattern decomposition, we consider generating $R(P_i)$ using $R(P_{i-1})$ and $R(p_i)$. Suppose γ is the ex-



(a) Vary N: $d = 5$ (b) Vary N: $d = 10$ (c) Vary N: $d = 100$ (d) Vary N: $d = 500$

Figure 4: The values of γ in different parameter combinations

pected number of matches in $R(P_i)$ that are generated from a certain match in $R(P_{i-1})$, we have:

$$|R(P_i)| = \gamma |R(P_{i-1})| \tag{11}$$

The value of γ depends on how p_i is joined with P_{i-1}. Suppose $p_i = \{(v, v'), (v, v'')\}$, in order to prove part 2, we need to prove the following S_1 and S_2 accordingly.

(S_1): We prove that $\gamma < 1$ when p_i is a strong TwinTwig with $v' \in V(P_{i-1})$ and $v'' \in V(P_{i-1})$. When $v \in V(P_{i-1})$, $\gamma < 1$ can be easily proved since no new node is added into $V(P_i)$. When $v \notin V(P_{i-1})$, suppose u' and u'' are arbitrary matches of v' and v'' respectively, we have:

$$\gamma = \mathbb{E}[\sum_{u \in V(G)} d(u')d(u)\rho \times d(u'')d(u)\rho]$$
$$= \mathbb{E}[d(u')d(u'')] \times \rho^2 \sum_{i=1}^{N} w_i^2$$

In order to calculate γ, we simplify the calculation of $\mathbb{E}[d(u')d(u'')]$ by only considering the relationship between u' and u''. There are two cases:

First, there is no edge between v' and v'' in P_{i-1}, and we consider that their matches, u' and u'', are independent. In this case, $\mathbb{E}[d(u')d(u'')] = \mathbb{E}[d(u')]\mathbb{E}[d(u'')] = d^2$. We have:

$$\gamma = d^2 \times \rho^2 \sum_{i=1}^{N} w_i^2 = \frac{\sum_{i=1}^{N} w_i^2}{N^2} \tag{12}$$

According to A_4, $w_i \leq d_{max} \leq \sqrt{N}$, therefore, $\gamma < \frac{d_{max}^2}{N} \leq 1$.

Second, there is an edge between v' and v'' in P_{i-1}. In this case, u' and u'' must have an edge in the data graph. Using the Bayes equation, we can derive the equation:

$$P(u' = u_i, u'' = u_j | u', u'' \text{ form an edge})$$
$$= \frac{P(u', u'' \text{ form an edge } | u' = u_i, u'' = u_j) \times P(u' = u_i, u'' = u_j)}{P(u', u'' \text{ form an edge})}$$
$$= \frac{P_{i,j} \times (1/N^2)}{2M/N^2} = \rho P_{i,j}$$

As a result, we have:

$$\mathbb{E}[d(u')d(u'')] = \sum_{i,j=1}^{N} \rho P_{i,j} w_i w_j$$
$$= \rho^2 (\sum_{i=1}^{N} w_i^2 \sum_{j=1}^{N} w_j^2) = \rho^2 (\sum_{i=1}^{N} w_i^2)^2$$

Therefore, γ can be calculated as:

$$\gamma = \rho^2 (\sum_{i=1}^{N} w_i^2)^2 \times \rho^2 \sum_{i=1}^{N} w_i^2 = \frac{(\sum_{i=1}^{N} w_i^2)^3}{(\sum_{i=1}^{N} w_i)^4} \tag{13}$$

It is hard to compute an upper bound for γ in this case. However, we show that $\gamma < 1$ for most real-world graphs. In order to do so, we vary β from 2.1 to 2.9, d from 5 to 500, and N from $10,000$ to $100,000,000$. Since γ increases with d_{max}, we set $d_{max} = \sqrt{N}$. With β, d, N, and d_{max}, we can generate $w_i (1 \leq i \leq N)$ via [37], and thus γ can be calculated via Eq. 13. The results are shown in Fig. 4, in which we can see that $\gamma < 1$ for all practical cases.

(S_2): We prove that $\gamma > 1$ when p_i is a non-strong TwinTwig with $u \in V(P_{i-1})$, $u' \notin V(P_{i-1})$, and $u'' \notin V(P_{i-1})$. In this situation, we have:

$$\gamma = \mathbb{E}[\sum_{u', u'' \in V(G)} d(u)d(u')\rho \times d(u)d(u'')\rho]$$
$$= \mathbb{E}[d(u)^2]\rho^2 \sum_{i,j=1}^{N} w_i w_j = \mathbb{E}[d(u)^2] = \sum_{i=1}^{N} w_i^2 / N \tag{14}$$

Obviously, $\gamma \geq \mathbb{E}[d(u)]^2 = d^2 > 1$. Now according to S_1 and S_2, part 2 is proved when p_i is a two-edge TwinTwig. When p_i only contains one edge, part 2 can be proved similarly.

According to Part 1 and Part 2, the instance optimality of the TwinTwig decomposition holds for a power-law random graph.

Optimal Decomposition. We show how to compute the optimal TwinTwig decomposition using A* for power-law random graph. Recall that Algorithm 2 is independent of the graph model. It is only required to compute $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$, which is a cost lower bound for any TwinTwig decomposition of P expanded from $\mathcal{D}_i$. In order to do so, we can simply set $\underline{\mathsf{cost}}(\mathcal{D}_i, P) = \mathsf{cost}(\mathcal{D}_i)$, where $\mathsf{cost}(\mathcal{D}_i)$ can be computed using Eq. 3, which depends on $|R(P_i)|$ and $|R(p_i)|$. Here, $|R(p_i)|$ can be precomputed, and $|R(P_i)|$ can be computed recursively using Eq. 11, where the value of each γ depends on how p_i is joined with P_{i-1}. Three typical cases for calculating γ are given in Eq. 12, Eq. 13, and Eq. 14, respectively. In this way, Algorithm 2 can be adopted to compute the optimal TwinTwig decomposition for the power-law random graph. The space and time complexities of the algorithm are the same as those shown in Lemma 5.3.

Optimization. In the three optimization strategies proposed in Section 6, workload skew reduction and early filtering are independent to the graph model. In order-aware cost reduction, reestimating $|R(p_i)|$ is also independent to the graph model. Therefore, we only discuss how to reestimate $|R(P_i)|$ in the power-law random graph. In order to do so, suppose for a partial pattern P_j with $j < i$, $|R(P_j)|$ has been accurately calculated, then for any future partial pattern P_i that is a supergraph of P_j, $|R(P_i)|$ can be calculated using Eq. 11 by considering adding TwinTwigs into P_j iteratively. Here how to compute γ after joining specific TwinTwigs is discussed in the above paragraph.

8. PERFORMANCE STUDIES

In this section, we show our experimental results. We deploy a cluster of up to 15 computing nodes including one master node and 14 slave nodes and we use 10 slave nodes by default. Each of the computing nodes has one 3.47GHz Intel Xeon CPU with 6 cores and 12GB memory running 64-bit Ubuntu Linux. We allocate a JVM heap space of 1024MB for each mapper and 2048MB for each reducer, and we allow at most 3 mappers and 3 reducers running concurrently in each machine. The block size in HDFS is set to be 128MB, the data replication factor of HDFS is set to be 3, and the I/O sort size is set to be 512MB.

Datasets. We use five real-world data graphs (see Table 1) for testing. Among them, *sk*, *lj*, *orkut*, and *fs* are downloaded from SNAP (http://snap.stanford.edu), *yt* is downloaded from KONECT (http://konect.uni-koblenz.de), and *uk* is downloaded from WEB (http://law.di.unimi.it).

Algorithms. We implement and compare seven algorithms:

- Edge: EdgeJoin (Section 4) with early filtering (Section 6.3).
- Mul: MultiwayJoin (Section 4).
- Star: StarJoin (Section 4) with early filtering (Section 6.3).
- TTBS: TwinTwigJoin (Section 5) without optimization.
- TTOA: TTBS + order-aware cost reduction (Section 6.1).
- TTLB: TTOA + workload skew reduction (Section 6.2).
- TT: TTLB + early filtering (Section 6.3).

dataset	name	$N = \|V\|$	$M = \|E\|$
as-skitter	*sk*	1,696,415	11,095,298
youtube	*yt*	3,223,589	12,223,774
live-journal	*lj*	4,847,571	42,851,237
com-orkut	*orkut*	3,072,441	117,185,083
uk-2002	*uk*	18,520,486	261,787,258
friendster	*fs*	65,608,366	1,806,067,135

Table 1: Datasets used in Experiments

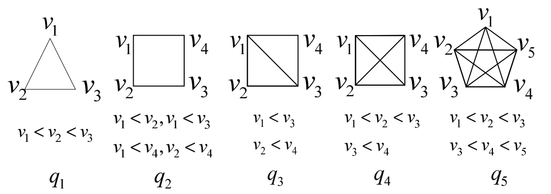


Figure 5: Queries

All algorithms are implemented using Hadoop (version 1.2.1) with Java 1.6. Note that the early filtering strategy (Section 6.3) is also applied in Edge and Star, and all the optimization strategies introduced in [1] are applied in Mul. We set the maximum running time to be 12 hours. If a test does not stop in the time limit, or fails due to out-of-memory exception, we denote the running time as INF. The time for computing the join plan using Algorithm 2 for TwinTwig decomposition is less than one second for all test cases, thus it is omitted in the total processing time.

Queries. The five queries denoted by q_1 to q_5 are illustrated in Fig. 5 with edge number varying from 3 to 10 and node number varying from 3 to 5. We show the vertex order for symmetry breaking under each query graph. Here, we only consider $n \leq 5$ for fair comparison, because when n is larger than 5, except for TT, all other algorithms cannot terminate in the time limit in most cases.

Exp-1: Vary Algorithms. In this experiment, we evaluate the performance of all seven algorithms using two query graphs q_3 and q_4 as representatives on the two datasets *yt* and *lj*. The experimental results are shown in Fig. 6. We also list the size of the output (see Table 2) generated by mappers and reducers in each round when we process q_4 on *lj*. Here we use "NA" to denote that the algorithm crashes due to out-of-memory exceptions, and use "-" to denote that no extra MapReduce round is needed. Note that we only present the results of the first three rounds for Edge which actually finishes in five rounds. The sizes of the output produced by TTLB and TTOA are the same, and thus we only show one of them. When evaluating q_3 on *yt*, we find that without early filtering, none of the algorithms can terminate in the time limit because *yt* contains a lot of high-degree nodes, thus we apply early filtering for both TTBS and TTOA in this case. The experimental results support our motivation to minimize the cost discussed in Section 5.2, as lower cost generally results in better performance.

As shown in Fig. 6, Mul fails in evaluating q_3 on *yt* and *lj*, and q_4 on *lj* due to out-of-memory exceptions. We analyze the reason below. Take the evaluation of q_4 on *lj* for example. Mul outputs 0.9 billion data, which is approximately 20 times larger than the size of the data graph. Since we need to use auxiliary data structures such as hash tables to index these data, each of which is represented by around 20 integers, leading to a 70GB memory consumption as a whole. However, we only configure 60GB memory for all reducers in the cluster (2GB per reducer for 30 reducers). Therefore, Mul runs out of memory.

Edge is slow and cannot finish in the time limit when evaluating q_3 on both *yt* and *lj*. This is because Edge often generates numerous partial results in early stages even after filtering. As shown in

m/r	Edge	Mul	Star	TTBS	TTLB	TT
map^1	0.09	0.90	10.20	2.77	1.36	0.57
reduce^1	0.29	NA	9.93	16.34	14.9	9.93
map^2	0.33	-	9.98	21.55	16.27	10.22
reduce^2	9.94	-	9.93	9.93	9.93	9.93
map^3	9.98	-	-	-	-	-
reduce^3	9.94	-	-	-	-	-
total	90.29	NA	40.07	50.59	42.49	30.67

Table 2: Size of Output for processing q_4 on lj (in billions)

Table 2, Edge has to deal with over 9.9 billion data from the third round, yet there are two more rounds to complete the task, in which more partial results are generated.

In most cases, Star is slower than TTBS, which demonstrates the instance optimality of TwinTwig decomposition in Theorem 5.1. However, TTBS spends much longer time than Star when evaluating q_4 on *yt*. This is because *yt* contains many high-degree nodes, and TTBS (without any optimization) can generate large number of partial results, while Star can avoid this issue by applying the early filtering strategy.

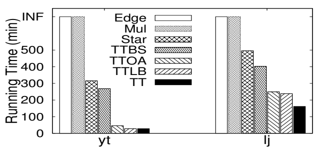
TTOA performs better than TTBS in all cases, which verifies the effectiveness of the order-aware cost reduction strategy, and TTLB outperforms TTOA in all cases, which is consistent with the analysis in Section 6.2. TT consistently outperforms all other algorithms for all test cases. Comparing TT to TTLB, we observe from Table 2 that TTLB generates 10 billion more data than TT, which shows the effectiveness of early filtering. In the rest of the experiments, we exclude the results of TTBS, TTOA, and TTLB, since their relative performances are similar to those shown in Fig. 6. Therefore, we focus on comparing Edge, Star, and Mul with our algorithm TT.

Exp-2: Vary Datasets. In this experiment, we test the algorithms on all the five datasets shown in Table 1 and show our results for query q_1 and q_4 for algorithms Edge, Mul, Star, and TT.

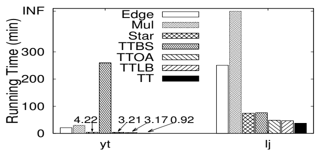
Fig. 7(a) shows the testing results for query q_1. Note that for q_1, star decomposition is the same as TwinTwig decomposition, hence Star has the same performance as TT, which outperforms Edge and Mul for over an order of magnitude. Generally, Mul performs slightly worse than Edge, except that Mul spends much longer time on *orkut*. This is because *orkut* contains too many edges, which results in a large number of edge duplications in Mul. Edge and Mul cannot handle large data graphs *uk* and *fs*.

The testing results for q_4 are shown in Fig. 7(b). TT is 5 times faster than Star on *orkut*, and is only 2 times faster than Star on *lj*. This is because that the larger the average degree of the data graph is, the better performance TT has over Star. The average degree of *orkut*, which is 76, is larger than that of *lj*, which is 28. Hence, such an experimental result is expected. Another interesting observation is that, when evaluating q_4, it takes longer time on *uk* than *fs*, while *uk* is much smaller than *fs*. The reason is that, *uk* is a web graph, which contains a lot of large cliques, since webpages in the same domain tend to link each other. On the contrary, *fs* is a social network, which contains fewer large cliques than a web graph.

Exp-3: Vary Queries. We evaluate all queries q_1 to q_5 in Fig. 5. The results are illustrated in Fig. 8(a) to Fig. 8(e) respectively. Note that Star is the same as TT when processing q_1 and q_2 since no node in q_1 and q_2 has degree larger than 2. Generally, the more complex the pattern graph is, the more time it takes to evaluate the query for all algorithms. TT performs the best in all test cases. Note that all the tests are conducted on *yt* and *lj* except for q_5, which is conducted on *yt* and *sk*. The reason is that, the number of results

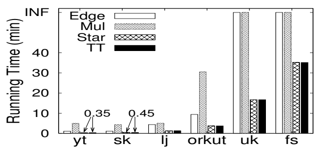


(a) Query q_3

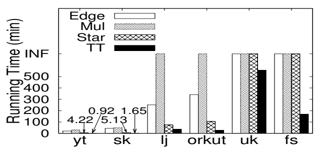


(b) Query q_4

Figure 6: Vary Algorithms

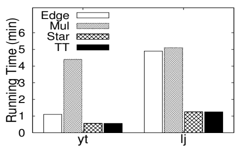


(a) Query q_1

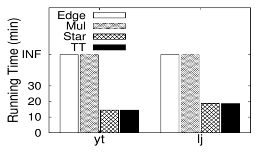


(b) Query q_4

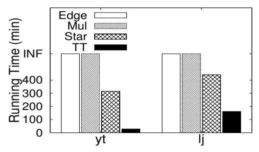
Figure 7: Vary Datasets



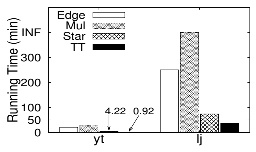
(a) Query q_1



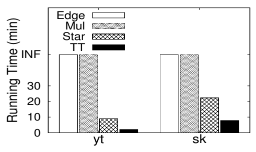
(b) Query q_2



(c) Query q_3

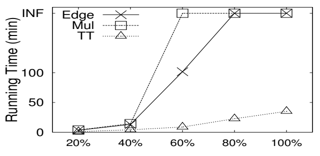


(d) Query q_4

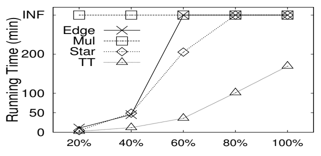


(e) Query q_5

Figure 8: Vary Queries

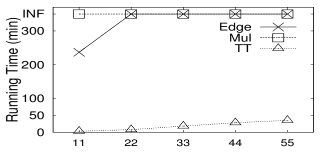


(a) Query q_1 on *fs*

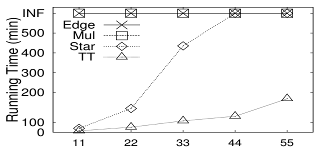


(b) Query q_4 on *fs*

Figure 9: Vary Graph Size



(a) Query q_1 on *fs*



(b) Query q_4 on *fs*

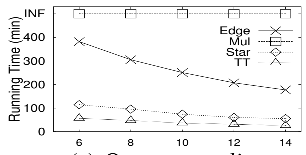
Figure 10: Vary Averge Degree

of q_5 on *lj* is over 400 billion, which surpasses the processing ability of our current cluster. However, we can scale to handle such a case by deploying more slave nodes.

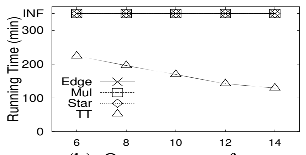
Exp-4: Vary Graph Size. We extract subgraphs of 20%, 40%, 60%, 80%, and 100% nodes from the original graph of *fs*, and test the algorithms using queries q_1 and q_4. The results are shown in Fig. 9(a) and Fig. 9(b) respectively. We omit the curve of Star in Fig. 9(a) since Star is the same as TT when evaluating q_1. When the graph size increases, the running time of Edge, Mul and Star grow much sharper than TT. When the graph size is over 80%, only TT can finish in the time limit. The testing results show the high scalability of our TT algorithm.

Exp-5: Vary Average Degree. We fix the set of nodes and randomly sample 20%, 40%, 60%, 80% and 100% edges from the original graph *fs* to generate graphs with average degrees from 11 to 55, and test the algorithms using queries q_1 and q_4. The results are shown in Fig. 10(a) and Fig. 10(b) respectively. We omit the curve of Star in Fig. 10(a) since Star is the same as TT when evaluating q_1. Edge and Mul fail at the very beginning. In Fig. 10(b), TT is 3, 5, 8 and $>$ 9 times faster than Star when the average degree varies from 11 to 55, which shows the advantage of TT for dense data graphs. The trend is consistent with our theoretical analysis in Section 5.

Exp-6: Vary Slave Nodes. In this experiment, we vary the number of slave nodes from 6 to 14, and evaluate our algorithms on the *lj* and *fs* dataset using query q_4. The testing results are shown in Fig. 11(a) and Fig. 11(b) respectively. As shown in Fig. 11(a), when the number of slave nodes increases, the processing time of all algorithms decreases, and the running time drops more sharply when the number of slave nodes is small. This is because that the increment of slave nodes, on the one hand, contributes to the performance improvement as workloads are more largely shared, on the other hand, introduces extra communication cost as more data transmissions are involved among slave nodes. As shown in Fig. 11(b), TT is the only algorithm that can compute the 4-clique on *fs* even when 14 slave nodes are deployed. We also make the tests using other queries when varying slave nodes. The curves are similar to those in Fig. 10 thus are omitted due to lack of space.



(a) Query q_4 on *lj*



(b) Query q_4 on *fs*

Figure 11: Vary Slave Nodes

Exp-7: Small-Degree Assumption. In this experiment, we show that the small-degree assumption A_4 is useful in practice. We call a vertex u with $d(u) > \sqrt{N}$ a *high-degree vertex*. For a data graph G, we create G^* by iteratively removing some edges of the high-degree vertices randomly until every vertex u in G has $d(u) \leq \sqrt{N}$. We denote $\mathcal{C}$ and $\mathcal{C}^*$ the cost (by Eq. 1) when evaluating a specific pattern in the graph G and G^*, respectively. And we denote $\alpha = \mathcal{C}^*/\mathcal{C}$ to show the ratio of the cost that is only related to G^* (in which our algorithm can guarantee instance optimality). In table Table 3, we show the value of α when evaluating q_1 and q_4 in the datasets *sk*, *yt* and *lj*, respectively. As we can see, the cost in G^* are actually the dominate part.

queries	*sk*	*yt*	*lj*
q_1	0.740	0.784	0.971
q_4	0.796	0.828	0.970

Table 3: The value of α

9. RELATED WORK

MapReduce Framework. MapReduce, introduced by Google, has attracted plenty of attentions is academia. A lot of researches focus on optimizing MapReduce framework. For example, cost analysis of MapReduce is given by Afrati et al. [2]. MapReduce classes are discussed by Karloff et al. [21] and Tao et al. [35]. Some other researches focus on solving specific queries in MapReduce. For example, Theta joins in MapReduce are discussed by Zhang et al. [41]. Multiway joins are optimized by Afrati et al. [3]. Duplication detection using MapReduce is proposed by Wang et al. [38].

Subgraph Matching. Most subgraph matching approaches work in a label-aware context, where vertices (and/or edges) are assigned labels in both data graph and query graph. For example, node labels in the neighborhood are utilized to filter unexpected candidates in [18] and [42]. In [17], the authors observe that a good matching order can significantly improve the performance of subgraph query. Inexact subgraph matching is also studied in [22], and [14]. Lee et al. [23] provide an in-depth comparison of subgraph isomorphism algorithms. Subgraph enumeration in a centralized environment is also studied in exact and approximate settings. The exact solutions including [8] and [16] are not scalable to handle large data graphs. The approximate solutions [5, 15, 43] only estimate the count rather than locate all the subgraph instances.

Subgraph Matching in Cloud. Due to the NP-hardness of the subgraph isomorphism problem, a lot of recent researches focus on solving subgraph matching in cloud. Zhao et al. [43] introduce a parallel color coding method for subgraph counting. Ma et al. [25] study inexact graph pattern matching based on graph simulation in a distributed environment. Gonzalez et al. report an experimental result on triangle counting in PowerGraph [19]. Recently, Sun et al. [33] propose a subgraph matching algorithm to utilize node filtering to handle labelled graphs in the Trinity memory cloud.

Subgraph Enumeration in MapReduce. MapReduce has been shown to be scalable to handle a lot of graph related problems, among which subgraph enumeration has attached lots of interests. Tsourakakis et al. [36] propose an approximate triangle counting algorithm using MapReduce. Suri et al. [34] introduce a MapReduce algorithm to compute exact triangle counting. Afrati et al. [1] propose multiway join in MapReduce to handle subgraph enumeration. Plantenga [29] introduces an edge join method in MapReduce which can be used for subgraph enumeration. Both [1] and [29] have been introduced in details in Section 4.

10. CONCLUSIONS

In this paper, we study scalable subgraph enumeration in MapReduce, considering that existing solutions for subgraph enumeration are not scalable enough to handle large graphs. We propose a new TwinTwigJoin algorithm based on a left-deep-join framework in MapReduce. In the Erdös-Rényi random-graph model, we show that under reasonable assumptions, TwinTwigJoin is instance optimal in the left-deep-join framework. An A*-based solution is given to compute the optimal join plan. We further improve our approach using three novel optimization strategies and extend our approach to handle the power-law random-graph model.We conduct extensive performance studies on real large graphs with up to billions of edges to demonstrate the effectiveness of our approach.

Acknowledgements. We would like to thank all the reviewers for their dedicated work, especially we want to thank Prof. Jeffrey D. Ullman for many insightful comments. Lu Qin is supported by ARC DE140100999. Xuemin Lin is supported by NSFC61232006, ARC DP120104168, ARC DP140103578, and ARC DP150102728. Lijun Chang is supported by ARC DE150100563.

11. REFERENCES

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *Proc. of ICDE'13*, 2013.

[2] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4), 2013.

[3] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9), 2011.

[4] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proc. of STOC '00*, 2000.

[5] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. In *Proc. of ISMB'08*, 2008.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[7] B. Bollobás. *Random graphs*. Springer, 1998.

[8] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1), 1985.

[9] F. R. K. Chung, L. Lu, and V. H. Vu. The spectra of random graphs with given expected degrees. *Internet Mathematics*, 1(3), 2003.

[10] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, Nov. 2009.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI'04*, 2004.

[12] P. Eades, X. Lin, and R. Tamassia. An algorithm for drawing a hierarchical graph. *International Journal of Computational Geometry & Applications*, 6(02):145–155, 1996.

[13] P. Erdos and A. Renyi. On the evolution of random graphs. In *Publ. Math. Inst. Hungary. Acad. Sci.*, 1960.

[14] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1), 2010.

[15] M. Gonen, D. Ron, and Y. Shavitt. Counting stars and other small subgraphs in sublinear time. In *Proc. of SODA'10*, 2010.

[16] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proc. of RECOMB'07*, 2007.

[17] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.

[18] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proc. of SIGMOD'08*, 2008.

[19] J.Gonzalez, Y.Low, H.Gu, D.Bickson, and C.Guestrin. Powergraph:distributed graph-parallel computation on natural graphs. In *Proc. of OSDI'12*, 2012.

[20] S. R. Kairam, D. J. Wang, and J. Leskovec. The life and death of online groups: Predicting group growth and longevity. In *Proc. of WSDM'12*, 2012.

[21] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proc. of SODA'10*, 2010.

[22] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. *PVLDB*, 6(3), 2013.

[23] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2012.

[24] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. In *Proc. of PAKDD'06*, 2006.

[25] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *WWW*, 2012.

[26] T. Milenkovic and N. Przulj. Uncovering biological network function via graphlet degree signatures. *Cancer Inform*, 6, 2008.

[27] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594), 2002.

[28] N.Shervashidze, S.Vishwanathan, T.Petri, K.Mehlhorn, and K.Borgwardt. Efficient graphlet kernels for large graph comparison. In *AISTATS*, 2009.

[29] T. Plantenga. Inexact subgraph isomorphism in mapreduce. *J. Parallel Distrib. Comput.*, 73(2), 2013.

[30] N. Przulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2), 2007.

[31] G. Rücker and C. Rücker. Substructure, subgraph, and walk counts as measures of the complexity of graphs and molecules. *Journal of Chemical Information and Computer Sciences*, 41(6), 2001.

[32] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing join orders. Technical report, 1993.

[33] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[34] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. of WWW'11*, 2011.

[35] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *Proc. of SIGMOD'13*, 2013.

[36] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proc. of KDD'09*, 2009.

[37] F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *COCOON'05*, pages 440–449, Berlin, Heidelberg, 2005. Springer-Verlag.

[38] C. Wang, J. Wang, X. Lin, W. Wang, H. Wang, H. Li, W. Tian, J. Xu, and R. Li. Mapdupreducer: detecting near duplicates over massive datasets. In *Proc. of SIGMOD'10*, pages 1119–1122, 2010.

[39] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9), 2012.

[40] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 6684(393), 1998.

[41] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11), 2012.

[42] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1-2), 2010.

[43] 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. In *Proc. of ICPP'10*, 2010.