Noname manuscript No.
(will be inserted by the editor)

Scalable Subgraph Enumeration in MapReduce: A Cost-Oriented Approach

Longbin Lai · Lu Qin · Xuemin Lin · Lijun Chang

Received: date / Accepted: date

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. We further discuss how our approach can be adapted to the power-law graphs. Three optimization strategies are explored to improve our algorithm. Ultimately, we utilize the compressed graph to further boost the algorithm. The compressed graph is computed by aggregating nodes that have the same neighbors into a compressed node. 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.

Keywords MapReduce · Subgraph Enumeration · Random Graph · Power-Law Graph

Longbin Lai
The University of New South Wales, Australia
E-mail: llai@cse.unsw.edu.au

Lu Qin
Centre for QCIS, University of Technology, Sydney, Australia
E-mail: lu.qin@uts.edu.au

Xuemin Lin
The University of New South Wales, Australia
E-mail: lxue@cse.unsw.edu.au

Lijun Chang
The University of New South Wales, Australia
E-mail: ljchang@cse.unsw.edu.au

1 Introduction

In this paper, we study subgraph enumeration, a fundamental problem in graph analysis. Given an undirected, unlabelled 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 many applications. For example, subgraph enumeration is used for network motif computing [26,4] to facilitate the design of large network from biochemistry, neurobiology, ecology, and bioinformatics. It is utilized to compute the graphlet kernels for large graph comparison [27,29] and property generalization for biological networks [25]. It is considered as a key operation for the synthesis of target structures in chemistry [32]. It is also adopted to illustrate the evolution of social networks [18] and to discover the information trend in recommendation networks [22]. In addition, as a special case of subgraph enumeration, triangle enumeration is a preliminary operation in cluster coefficient calculation [40] and community detection [39].

Motivation. Enumerating subgraphs in a big data graph, despite its varied applications, is extremely challenging for two reasons. 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-complete. 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 [7,14] are not scalable to handle big graphs. Some other studies try to find approximate solutions [4,13,42] to reduce the computational cost, however, they only estimate the count of the matched subgraphs rather than locate all the subgraph instances.

MapReduce [10], as one of the most popular parallel computing paradigms for big data processing, has been widely used in both industry and academia. MapReduce features high scalability, reliability, fault-tolerance and the easy-to-use programming interfaces. In the literature, two existing approaches focus on subgraph enumeration using MapReduce, namely, edge-based join [28] and multiway join [1].

In edge-based join [28], the pattern graph is decomposed into an ordered list of edges. The algorithm proceeds in multiple MapReduce rounds, each of which 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. 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.

We propose a new approach for subgraph enumeration using MapReduce in this paper considering the drawbacks of edge-based join and multiway join. We introduce a two-way-join framework that generalizes the edge-based join to allow joining a star (a tree of depth one) instead of a single edge in each round. However, joining a star is sometimes inefficient as well. Thus, we propose the TwinTwigJoin which uses a TwinTwig (an edge or two incident edges of a node) as the basic join element 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 as well as a variant of Erdös Rényi random-graph model, we show that TwinTwigJoin can ensure instance optimality in the two-way-join framework. Second, the simple structure of a TwinTwig makes it easy to devise an optimal join plan based on the A* algorithm. Third, a lot of 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 first deliver the result of instance optimality by assuming that the data graph is a random graph. This not only provides the theoretic evidence of why the algorithm presented in the paper is sound but also gives the foundation of our analysis of power-law graphs. Later, we extend the results to power-law graphs with the aim to cover real-life large graphs, such as social networks, web graphs, and protein-protein interaction networks.

We further improve the performance of the algorithm by exploiting the node-equivalence relationships. A set of nodes that have the *same neighborhoods* are considered as **equivalent nodes** and aggregated into one **compressed node**, which transforms the original graph into a **compressed graph**. Ren et al. applied the technique in [31] to boost the subgraph matching. However, their centralized algorithm cannot scale to web-scale real graphs. In this paper, we propose **non-trivial** MapReduce algorithms to solve subgraph enumeration regarding the compressed graph via two challenging tasks:

- (1) Construct the compressed graph. In the distributed context, we can only access to the neighbors while processing each node. Therefore, we need to design the algorithm carefully to ensure that the algorithm is correct and at the same time, it is computationally efficient.
- (2) Query the compressed graph. We cannot directly follow the backtracking algorithm proposed by [31]. On the other side, we cannot trivially extend the proposed join algorithm due to the involvement of compressed nodes.

To solve task (1), we first identify three kinds of compressed nodes. Then we use those nodes to bind the compressed edges and construct the compressed graph. The algorithm has linear communication cost, and hence can scale to web-scale real graphs. Considering the properties of the compressed nodes, we carefully devise a new algorithm based on the join framework to tackle task (2). We formally prove the correctness of all the proposed algorithms.

Contributions. We make the following contributions in this paper.

(1) A left-deep-join framework to allow joining multiple edges in each round. We introduce a framework based on left-deep join for subgraph enumeration in MapReduce (Section 3), which generalizes the edge-based join to allow multiple edges (in forms 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 basic join element 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. We further develop an A*-based algorithm to compute the optimal join plan for TwinTwigJoin 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. It is worth noting that we propose a comprehensive cost model for subgraph enumeration for the first time. According to the proposed cost model, our subgraph enumeration algorithm is optimal.

(3) Extension to power-law graphs. We show how our algorithms and theoretical results can be adapted to the power-law graph model in Section 6.

(4) Three optimization strategies to further improve the algorithm. We explore three optimization strategies in Section 7, 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. This is accomplished by partitioning the neighbors of high-degree nodes into multiple machines. Early filtering makes use of the free memory to further filter invalid partial results in early stages of the algorithm.

(5) Compressed graph based on node equivalence. We develop non-trivial MapReduce algorithms to construct the compressed graph by aggregating the equivalent nodes into a compressed node. We show that the communication cost of the algorithm is linear to the graph size, thus it can scale to large graphs. We further devise an algorithm based on the join framework to process subgraph enumeration on the compressed graph. We show the correctness of all proposed algorithms (Section 8).

(6) Extensive performance studies using web-scale real graphs. 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 achieves high scalability and outperforms all other state-of-the-art algorithms in all datasets (Section 9).

Outline. Section 2 presents the preliminaries and gives the formal problem definition. Section 3 shows the join framework for solving subgraph enumeration. Section 4 introduces three state-of-the-art algorithms for subgraph enumeration in MapReduce. Section 5 studies a new TwinTwigJoin algorithm, proves its instance optimality using the Erdös-Rényi random-graph model, and provides an optimal join plan based on the A* algorithm. Section 6 shows how our algorithm can be extended to deal with the power-law graph model. Section 7 explores three optimization strategies to further optimize the TwinTwigJoin algorithm. Section 8 studies how to construct the compressed graph and how our algorithm can be adapted to process queries over the hyper graph. Section 9 evaluates all introduced algorithms using extensive experiments. Section 10 reviews the related work, and Section 11 concludes the paper. To enhance readability, we have included all proofs of lemmas and theorems in the appendix.

2 Problem Definition

Subgraph Enumeration. We model a *data graph* as an undirected and unlabeled graph $G = (V(G), E(G))$, where $V(G)$ represents the set of nodes and $E(G)$ 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 $\{u_1, u_2, \ldots, u_N\}$ 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 $P = (V(P), E(P))$, 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 $\{v_1, v_2, \ldots, v_n\}$ 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 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 $f = (u_{k_1}, u_{k_2}, \ldots, u_{k_n})$ to denote the match f, i.e., $f(v_i) = u_{k_i}$ for any $1 \leq i \leq n$.

Definition 2 (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 $|E(g_i)| = |E(g_j)|$.

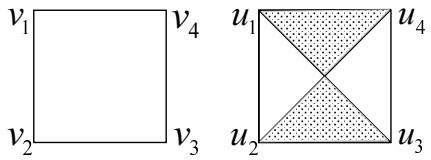


Fig. 1 Pattern Graph P (Left) and Data Graph G (Right).

Definition 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 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 $|\mathcal{A}(P)| \geq 1$.

Example 1 Fig. 1 shows a pattern graph P which is a square, and a data graph G with 4 nodes and 6 edges. We can find the following three subgraphs of G that is isomorphic to P: (u_1, u_2, u_3, u_4) (the peripheral square), (u_1, u_3, u_2, u_4) (the shadowed part), and (u_1, u_2, u_4, u_3) (the white part).

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 3) using MapReduce.

3 Algorithm Framework

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

```
Input  : G : The data graph,
         P : The pattern graph.
Output : R(P_t): All Matches of P_t in G.
function SubgraphEnum (G, P)
compute a graph decomposition {p_0, p_1, ..., p_t} of P;
for i = 1 to t do
    R(P_i) ← R(P_{i-1}) ⋈ R(p_i); (using map^i and reduce^i)
return R(P_t);
function map^i( key: ∅; value: Either a match
f ∈ R(P_{i-1}) when i > 1 or (u, N(u)) for a node
u ∈ V(G) )
{v_{k_1}, v_{k_2}, ..., v_{k_l}} ← V(P_{i-1}) ∩ V(p_i);
if i = 1 then
    G_u ← a graph formed by edges (u, v) for v ∈ N(u);
    R_u(P_0) ← all matches of P_0 in G_u;
    forall the match f ∈ R_u(P_0) do
        output ((f(v_{k_1}), f(v_{k_2}), ..., f(v_{k_l})); f);
if Value is a match f ∈ R(P_{i-1}) then
    output ((f(v_{k_1}), f(v_{k_2}), ..., f(v_{k_l})); f);
else
    G_u ← a graph formed by edges (u, v) for v ∈ N(u);
    R_u(p_i) ← all matches of p_i in G_u;
    forall the match h ∈ R_u(p_i) do
        output ((h(v_{k_1}), h(v_{k_2}), ..., h(v_{k_l})); h);
function reduce^i( key:r = (u_{k_1}, u_{k_2}, ..., u_{k_l}); value:
F = {f_1, f_2, ... }, H = {h_1, h_2, ... } )
forall the (f, h) ∈ (F × H) s.t. (f − r) ∩ (h − r) = ∅ do
    output (∅; f ∪ h);
```

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 processed 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 5 (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 $(0 \leq i \leq t)$ is a **star** (a tree of depth 1), and $V(p_i) \cap \bigcup_{0 \leq j < i} V(p_j) \neq \emptyset$ $(i \neq 0)$.

Definition 6 (Partial Pattern P_i) Given a pattern decomposition $\{p_0, p_1, \ldots, p_t\}$ of P, a *partial pattern* P_i $(0 \leq i \leq t)$ is a subgraph of P, such that $V(P_i) = \bigcup_{0 \leq j \leq i} V(p_j)$ and $E(P_i) = \bigcup_{0 \leq j \leq i} E(p_j)$. 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 $0 \leq i \leq t$.

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

Definition 7 (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 2). Then the algorithm is processed in t MapReduce rounds. Each round (lines 3-4) 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 $1 \leq i \leq t$. Each join operation is processed using MapReduce with map^i and reduce^i.

(Function map^i): The function map^i is shown in lines 6-19 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 6). 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 7). 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 8-12). 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 (line 14). Otherwise, we compute the matches of p_i associated with u, $R_u(p_i)$, as we do when we compute P_0 (lines 15-19).

(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 1)(lines 20-22).

Discussion. In the map^i phase of Algorithm 1, we need to compute $R(P_0)$ for $i = 1$ (lines 8-12) and $R(p_i)$ for $1 \leq i \leq t$ (lines 15-19) in G. Note that $R(P_0) = R(p_0)$, thus overall we need to compute $R(p_i)$ for $0 \leq i \leq t$ 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 node combinations from $\mathcal{N}(u)$.

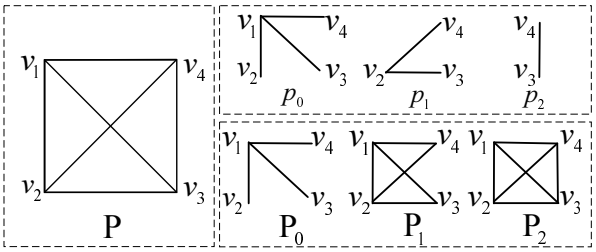


Fig. 2 The pattern decomposition and the corresponding partial patterns.

Example 2 In Fig. 2, 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.

Left-Deep vs. Bushy. 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 [34], 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.

Induced match. We show that Algorithm 1 can be adpated to *induced match*. The induced match, compared to Definiton 1, requires another *inducing constraint*, that is, for two pattern nodes v_i and v_j, if $(v_i, v_j) \notin E(P)$, then $(f(v_i), f(v_j)) \notin E(G)$. Algorithm 1 can be adapted to computing the induced match by filtering the (partial) matches that do not satisfy the inducing constraint in each step. Consider the pattern graph P and its partial pattern P_i whose results $R(P_i)$ are computed in the i^{th} step. If P_i does not include nodes that are not in P_{i-1}, then there needs no filtering. Otherwise, denote V_i^+ as the set of nodes added by P_i. We define the *suspected edges* for P_i as $E_i^s = \{(v, v') \mid v \in V_i^+ \wedge v' \in V(P_i) \wedge (v, v') \notin E(P)\}$, and the *suspected node cover* V_i^s as the **minimum** set of nodes in P_i that covers E_i^s. Let $V_i^s = \{v_{s_1}, v_{s_2}, \ldots, v_{s_i}\}$. Then we use s_i rounds of MapReduce to filter non-induced matches. Suppose $\tilde{R}^{(j-1)}(P_i)$ is the set of in-

duced matches after the $(j-1)^{th}$-round filtering ($0 < j <= s_i$ and note that $\tilde{R}^{(0)}(P_i) = R(P_i)$. In the j^{th} round, we apply the following filtering procedure for each $f \in \tilde{R}^{(j-1)}(P_i)$. Let $\tilde{u} = f(v_{s_j})$. We produce the key-value pair $(\tilde{u}; f)$ in the map phrase and it will reach the machine that the data node $\tilde{u}$ is stored (as $(\tilde{u}; \mathcal{N}(\tilde{u}))$). In the reduce phrase, we will filter f if $(v_{s_j}, v') \notin E(P) \wedge (\tilde{u}, f(v')) \in E(G)$. In this way, we are able to filter non-induced matches as early as possible and finally obtain the induced matches.

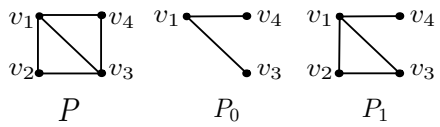


Fig. 3 A pattern graph P and its two consecutive partial patterns P_0 and P_1.

Example 3 Fig. 3 presents a pattern graph P, and its two consecutive partial patterns P_0 and P_1. Obviously, the node set added by P_1, $V_1^+ = \{v_2\}$, and the suspected edge set $E_1^s = \{(v_2, v_4)\}$ as $v_2 \in V_1^+$ and $(v_2, v_4) \notin E(P)$. As a result, we have $V_1^s = \{v_2\}$ to cover E_1^s. When we obtain a match of P_1, $f = (u_1, u_2, u_3, u_4)$, we will process the filtering by mapping f to where u_2 is stored (simple setting u_2 as the key in the map function). If $u_4 \in \mathcal{N}(u_2)$, f shall be filtered due to the violation of the inducing constraint, otherwise it shall be preserved.

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 using only one MapReduce round by duplicating edges in the data graph G. In the following, we introduce EdgeJoin and StarJoin in Section 4.1 and introduce MultiwayJoin in Section 4.2.

4.1 Left-Deep Join

Algorithm EdgeJoin. The EdgeJoin Algorithm is proposed by Plantenga [28]. In EdgeJoin, each pattern graph P is decomposed into $\{p_0, p_1, \cdots, p_t\}$ where each p_i is an edge in $E(P)$. Thus, we have $t = m - 1$. The EdgeJoin Algorithm has two drawbacks. Firstly, it may generate a large number of intermediate matches. Secondly, it needs $m - 1$ MapReduce rounds, which is large. We explain the two drawbacks using the following example.

Example 4 For the square given in Example 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 needs 3 MapReduce rounds, and the partial pattern P_3 is a path of length 3, which may result in a large number of intermediate matches in $R(P_3)$ comparing to $|R(P)|$, i.e., $|R(P_3)| >> |R(P)|$. 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 1 MapReduce round, and the size of the intermediate matches is not large comparing to the size of the final result $R(P)$.

Algorithm StarJoin. The StarJoin algorithm decomposes the pattern graph into a list of stars. The star decomposition strategy is proposed by Sun et al. [35]. Given a pattern graph P, and a node $v \in V(P)$, denote $star(v)$ the star rooted at v with $\mathcal{N}(v)$ as its child nodes. A star decomposition of P is defined as follows.

Definition 8 (Star Decomposition) Given a pattern graph P, a *star decomposition* is a decomposition $\{p_0, p_1, \cdots, p_t\}$ of P, such that there exists $\{v_{k_0}, v_{k_2}, \cdots, v_{k_t}\} \subseteq V(P)$ with $p_0 = star(v_{k_0})$, and $p_i = star(v_{k_i}) \setminus P_{i-1}$ for any $1 \leq i \leq t$.

Comparing to EdgeJoin, StarJoin can largely reduce the number of MapReduce rounds, however, StarJoin still suffers from the scalability issue due to the large number of intermediate matches generated when evaluating a star with a large degree. We explain the drawback using the following example.

Example 5 Fig. 2 shows an example of star decomposition for a 4-clique pattern graph P, in which p_0 is a star with degree 3. In a real social network such as Twitter, it is very common for a person to have more than 10,000 followers. As a result, matching the root of p_0 to this single person alone will produce more than 10^{12} intermediate matches.

4.2 Multiway Join

The MultiwayJoin algorithm was 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 [36] 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 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 9 (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 makes use of more structural information of the pattern graph to reduce the size of the intermediate results. Compared to StarJoin, TwinTwigJoin avoids 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 10 (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:

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

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

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

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

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

For any $1 \leq i \leq t$, 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 [11], denoted ER model, and the power-law random graph model [3], 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 6.

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.

Lemma 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 2 *Consider an SDEC pattern graph P, and one of its strong TwinTwig decompositions, $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$. For any partial pattern P_i $(1 \leq i \leq t)$, we have*

$$|R(P_i)| \leq |R(P_{i-1})| \times \frac{(2M)^2}{N^3} \leq |R(p_0)| \times (\frac{(2M)^2}{N^3})^i.$$

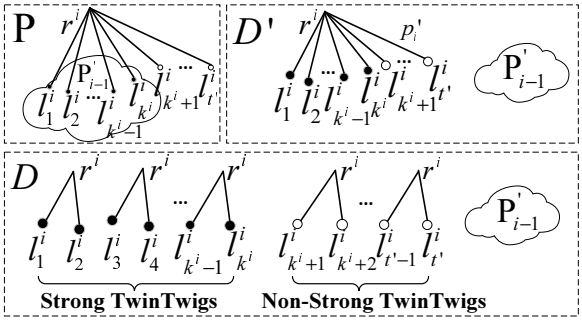


Fig. 4 Constructing the TwinTwig decomposition $\mathcal{D}$ based on a certain star decomposition $\mathcal{D}'$.

Corollary 1 *Consider an SDEC pattern graph P, and one of its strong TwinTwig decomposition, $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$. Under the assumption A_3, for any partial pattern P_i $(1 \leq i \leq t)$, we have*

$$|R(P_i)| \leq |R(P_{i-1})| \leq \ldots \leq |R(P_0)| = |R(p_0)|.$$

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 $(0 \leq i \leq t')$ is a star, we can construct a corresponding TwinTwig decomposition $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$ with $\mathsf{cost}(\mathcal{D}) \leq \Theta(\mathsf{cost}(\mathcal{D}'))$.

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 $1 \leq j \leq k_i$, $l^i_j \in V(P'_{i-1})$, and if $k_i < j \leq t'_i$, $l^i_j \notin V(P'_{i-1})$. $\mathcal{D}^i = \{p^i_1, p^i_2, \ldots, p^i_{t_i}\}$ is constructed as follows:

- If t'_i is an even number, then $t_i = \frac{t'_i}{2}$, and p^i_j $(1 \leq j \leq t_i)$ 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 $(1 \leq j \leq t_i - 1)$ 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 $0 \leq i \leq t'$, 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. 4.

We show the instance optimality of a general TwinTwig decomposition in the following theorem.

Theorem 1 *Consider a pattern decomposition $\mathcal{D}' = \{p'_0, p'_1, \ldots, p'_{t'}\}$ where each p'_i $(0 \leq i \leq t')$ is a star.*

Let $\mathcal{D}$ be the TwinTwig *decomposition constructed based on $\mathcal{D}'$ using the above method. Under the assumption A_3, we have* $\mathsf{cost}(\mathcal{D}) \leq \Theta(\mathsf{cost}(\mathcal{D}'))$.

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 6), 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 matches of any connected pattern graph P with m edges and n nodes, according to Lemma 1, we have

$$\mathsf{card}(m, n) = (2M)^m / N^{2m-n}. \tag{5}$$

In the dynamic-programming 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

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

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

$$\begin{aligned}\underline{\mathsf{cost}}(\mathcal{D}_i, P) = \mathsf{cost}(\mathcal{D}_i) + \\ \underline{\Delta\mathsf{cost}}(P_i, |E(P)| - |E(P_i)|, |V(P)| - |V(P_i|).\end{aligned} \tag{6}$$

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 11), (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 8), or (4) pushes the new entry into $\mathcal{H}$ if the corresponding partial pattern is not in $\mathcal{H}$ (line 10). 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 12).

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

```
   Input  : G, The data graph,
            P, The pattern graph.
   Output : The optimal decomposition of P.
1  H ← ∅;
2  forall the TwinTwig p in P do
3      H.push((p, {p}, cost({p}, P)));
4  (P', D', cost(D', P)) ← H.pop();
5  while P' ≠ P do
6      forall the TwinTwig p with V(p) ∩ V(P') ≠ ∅ and
       E(p) ∩ E(P') = ∅ do
7          if H.find(P' ∪ p) ≠ ∅ then
8              H.update(P' ∪ p, D' ∪ {p}, cost(D' ∪ {p}, P));
9          else
10             H.push((P' ∪ p, D' ∪ {p}, cost(D' ∪ {p}, P)));
11     (P', D', cost(D', P)) ← H.pop();
12 return D';
```

Lemma 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 $\overline{d} = \max_{v \in V(P)} d(v)$.*

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

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 symmetry-breaking techniques introduced in [14]. 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. Symmetry breaking is then performed by assigning a **partial order** (defined by $<$) among some pairs of nodes in the pattern graph P. Given such a partial order, a match is redefined as follows:

Definition 11 (Match) A *match* f from a pattern graph P to a data graph G is a mapping from $V(P)$ to $V(G)$ that satisfies:

- *(Conflict Freedom)* The same as that in Definiton 1.
- *(Structure Preservation)* The same as that in Definiton 1.
- *(Order Preservation)* For any pair of nodes $v_i \in V(P)$ and $v_j \in V(P)$, if $v_i < v_j$, then $f(v_i) \prec f(v_j)$.

Compared to Definiton 1, a new *order-preservation* constraint is added in the new definition of a *match*.

Example 6 The square given in Example 1 has 8 automorphisms. Thus, each result subgraph will be duplicated 8 times using Algorithm 1. For example, the 8 matches (u_1, u_2, u_3, u_4), (u_2, u_3, u_4, u_1), (u_3, u_4, u_1, u_2), (u_4, u_1, u_2, u_3), (u_4, u_3, u_2, u_1), (u_3, u_2, u_1, u_4), (u_2, u_1, u_4, u_3), and (u_1, u_4, u_3, u_2) all represent the same subgraph with 4 edges (u_1, u_2), (u_2, u_3), (u_3, u_4), and (u_4, u_1). Suppose $u_1 \prec u_2 \prec u_3 \prec u_4$, by defining a partial order: $v_1 < v_2$, $v_1 < v_3$, $v_1 < v_4$, and $v_2 < v_4$ in P, only one match (u_1, u_2, u_3, u_4) is left.

Algorithm 1 can be extended to handle the partial order as follows: In the map^i phase, when computing $R(p_i)$ (line 10, line 17), we make sure that each match satisfies the *order preservation* constraint. In the reduce^i phase, in line 22, we only output those $f \cup h$ that satisfy the *order preservation* constraint. In Section 7.1, we will discuss how to use the partial order to further optimize the pattern decomposition.

6 Handling Power-Law Graphs

In this section, we will show how to adapt TwinTwigJoin to the power-law graphs.

We model the data graph G of N nodes and M edges as a power-law random graph according to [3]. We consider a non-increasing degree sequence $\{w_1, w_2, \ldots, w_N\}$ 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 [8,9]. 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 show the intermediate results from the nodes with degree $\le \sqrt{N}$ play a dominant role in the total intermediate results.

Instance Optimality. In order to show the instance optimality of TwinTwigJoin in power-law graphs, we prove that Theorem 1 holds in a power-law random graph model under the small-degree assumption A_4. The detailed proof can be found in the appendix.

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 3.

Optimization. In the three optimization strategies proposed in Section 7, 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.

7 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.

7.1 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, $|R(p_1)| = |R(p_2)| = |R(p_3)| = \Theta(N^2)$.

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 12 (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 $V(G)$.

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 $v_2 < v_1 < v$.

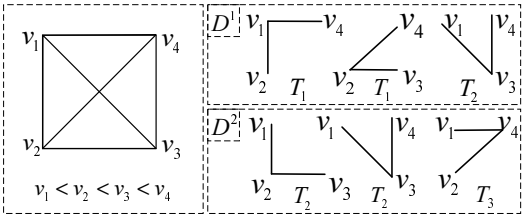


Fig. 5 The order-aware decomposition of a 4-Clique.

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 [7]. 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 7 Fig. 5 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 $\mathcal{D}^2$.

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. 6) 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. 8, $|R(P_i)|$ can be calculated as:

$$|R(P_i)| = |R(P_j)| \times (\frac{2M}{N^2})^{|E(P_i)|-|E(P_j)|} \times N^{|V(P_i)|-|V(P_j)|} \quad (7)$$

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 $0 \leq j < i$ is replaced by the accurate value. In this way, the partial order is involved in Algorithm 1.

7.2 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: $\mathcal{N}(u) = \{\mathcal{N}_1(u), \mathcal{N}_2(u), \ldots, \mathcal{N}_\beta(u)\}$. 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 $u_2 \in \mathcal{N}_j(u)$ $(i \neq j)$ 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.

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

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 $\Theta(\sqrt{\lambda} \cdot |\mathcal{N}(u)|)$.

7.3 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 8 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.

8 Compressed Graph

By aggregating nodes that have the same neighbors into a *compressed node*, we construct a *compressed graph*, upon which the performance of subgraph enumeration

is further improved. In this section, we will introduce the MapReduce algorithm that correctly constructs the compressed graph, and show that the algorithm has linear communication cost, and hence can scale to web-scale real graphs. We will then discuss how to utilize the join framework (Algorithm 1) to correctly process the query upon the compressed graph.

To start, we define the *Node Equivalence*, *Compressed Node* and *Compressed Graph*.

Definition 13 (**Node Equivalence**) Given two nodes u_i and u_j in the data graph G, we say u_i is equivalent to u_j, denoted $u_i \simeq u_j$, if and only if $\mathcal{N}(u_i) \setminus \{u_j\} = \mathcal{N}(u_j) \setminus \{u_i\}$.

Given the node-equivalence relation, we partition the data nodes into a set of equivalence classes.

Definition 14 (**Compressed node**) Given a data graph G, the *compressed node* regarding u, denoted $\mathcal{S}(u)$, represents a set of nodes in $V(G)$ that are equivalent to u (u included). We denote $|\mathcal{S}(u)|$ as the size of the compressed node. A compressed node $\mathcal{S}(u)$ is **trivial** if $\mathcal{S}(u) = \{u\}$.

$\mathcal{S}(u)$ represents an equivalence class of u w.r.t. the node-equivalence relation, and it is clear that $\mathcal{S}(u) = \mathcal{S}(u')$ for any $u' \in \mathcal{S}(u)$. We call the node with the minimum identity in $\mathcal{S}(u)$ the *representative data node* of $\mathcal{S}(u)$, denoted as $r_{\mathcal{S}(u)}$. For the ease of presentation, we also use $\mathcal{S}_{id(r_{\mathcal{S}(u)})}$ to denote $\mathcal{S}(u)$. It is worth noting that the nodes inside a non-trivial compressed node $\mathcal{S}$ either form a clique (mutually connected) or an independent set (mutually disconnected). We use a field, $\mathcal{S}.clique$, to distinguish the two cases. Specifically, if the nodes in $\mathcal{S}$ form a clique, we call $\mathcal{S}$ a *clique compressed node* and set $\mathcal{S}.clique$ = true, otherwise, we call it an *independent compressed node* and set $\mathcal{S}.clique$ = false. We also set $\mathcal{S}.clique$ = false if $\mathcal{S}$ is trivial.

Definition 15 (**Compressed Graph**) Given a data graph $G = (V(G), E(G))$, the compressed graph corresponding to G is a graph $G^* = (V(G^*), E(G^*))$, such that,

- $V(G^*) = \{\mathcal{S}(u) \mid u \in V(G)\}$,
- $E(G^*) = \{(\mathcal{S}(u), \mathcal{S}(u')) \mid \mathcal{S}(u), \mathcal{S}(u') \in V(G^*) \wedge \mathcal{S}(u) \neq \mathcal{S}(u') \wedge (u, u') \in E(G)\}$. Each edge in $E(G^*)$ is called a **compressed edge**.

We define the **compressed neighbors** of a compressed node $\mathcal{S}$ in G^* as $\mathcal{N}^*(\mathcal{S}) = \{\mathcal{S}' \mid (\mathcal{S}, \mathcal{S}') \in E(G^*)\}$.

Definition 16 (**Compressed Node Order** $\prec^*$) Given two compressed nodes $\mathcal{S}(u)$ and $\mathcal{S}(u')$ in $V(G^*)$, we say $\mathcal{S}(u) \prec^* \mathcal{S}(u')$, if and only if $r_{\mathcal{S}(u)} \prec r_{\mathcal{S}(u')}$ (Definiton 12).

We then revise the total order $\prec$ (Definiton 12) as $\prec'$ in the data graph.

Definition 17 (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:

- $\mathcal{S}(u_i) \prec^* \mathcal{S}(u_j)$,
- $\mathcal{S}(u_i) = \mathcal{S}(u_j)$ and $id(u_i) < id(u_j)$.

Example 9 In the data graph G presented in Fig. 6, we have $u_1 \simeq u_2 \simeq u_3$ and $u_5 \simeq u_7$. Hence the compressed nodes are, $\mathcal{S}_1 = \mathcal{S}(u_1) = \{u_1, u_2, u_3\}$, $\mathcal{S}_4 = \mathcal{S}(u_4) = \{u_4\}$, $\mathcal{S}_5 = \mathcal{S}(u_5) = \{u_5, u_7\}$, $\mathcal{S}_6 = \mathcal{S}(u_6) = \{u_6\}$ and $\mathcal{S}_8 = \mathcal{S}(u_8) = \{u_8\}$. Among the compressed nodes, we have $h_1.clique$ = true, and false for the others. We then construct the compressed graph G^* by connecting the compressed nodes. For example, $(\mathcal{S}_1, \mathcal{S}_4) \in E(G^*)$ as $(u_1, u_4) \in E(G)$, and $(\mathcal{S}_5, \mathcal{S}_6) \in E(G^*)$ as $(u_5, u_6) \in E(G)$. It is clear that $\mathcal{S}_5 \prec^* \mathcal{S}_6 \prec^* \mathcal{S}_1 \prec^* \mathcal{S}_8 \prec^* \mathcal{S}_4$ regarding Definiton 16.

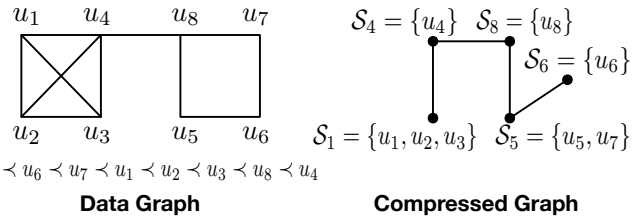


Fig. 6 The Compressed node and compressed graph of the Given Data Graph.

8.1 Constructing the Compressed Graph

We show how to construct the compressed graph corresponding to a given data graph G using MapReduce. We divide the process into two steps, namely *Compressed-Node Generation* and *Compressed-Edge Binding*.

Compressed-Node Generation. Given a data graph G and the corresponding compressed graph G^*, there are three cases for a compressed node $\mathcal{S} \in V(G^*)$:

- If $\mathcal{S}$ is a clique compressed node, then $\forall u, u' \in \mathcal{S}$, $\mathcal{N}[u] = \mathcal{N}[u']$, where $\mathcal{N}[u] = \mathcal{N}(u) \cup \{u\}$ is the **closured neighbors** of u;
- If $\mathcal{S}$ is an independent compressed node, then $\forall u, u' \in \mathcal{S}, \mathcal{N}(u) = \mathcal{N}(u')$;
- If $\mathcal{S}$ is a trivial compressed node, then $\mathcal{S} = \{u\}$ for some $u \in V(G)$.

Intuitively, to compute the compressed nodes is to aggregate the nodes that have the same (closured) neighbors. Algorithm 3 describes the detailed algorithm, in which we will attach the symbols "$\boxtimes$", "$\because$" and "$\times$" to indicate what kind of compressed node is being processed, as shown in Table 1.

Algorithm 3: ComprNodeGen(data graph G)

Input : G, The data graph,
Output : $V(G^*)$, The compressed node set.

1 **function** map^1(**key**: u; **value**: $\mathcal{N}(u)$)
2 **if** $d(u) \leq \sqrt{M}$ **then** output $(\mathcal{N}[u]; u)$;
3 **else** output $(\{\$\}; u)$;

4 **function** reduce^1(**key**: U; **value**: $S = \{u_{s_1}, u_{s_2}, \ldots, u_{s_k}\}$, where $u_{s_1} \prec u_{s_2}, \cdots \prec u_{s_k}$)
5 **if** $V = \{\$\}$ **then**
6 **foreach** $u \in S$ **do**
7 output $(u; (\therefore, \{u\}))$;
8 **else**
9 **if** $|S| > 1$ **then**
10 output $(u_{s_1}; (\boxtimes, S))$;
11 **else** output $(u_{s_1}; (\times, S = \{u_{s_1}\}))$;

12 **function** map^2(**key**: u; **value**: $\mathcal{N}(u)$)
13 **if** $d(u) \leq \sqrt{M}$ **then** output $(\mathcal{N}(u); u)$;

14 **function** reduce^2(**key**: U; **value**: $S = \{u_{s_1}, u_{s_2}, \ldots, u_{s_k}\}$, where $u_{s_1} \prec u_{s_2} \cdots \prec u_{s_k}$)
15 **if** $|S| > 1$ **then**
16 output $(u_{s_1}; (\therefore, S))$;
17 **else** output $(u_{s_1}; (\times, S = \{u_{s_1}\}))$;

18 **function** map^3(**key**: u; **value**: (x, S), where $x \in \{\therefore, \boxtimes, \times\}$, $u \in S$)
// map^3 reads both the outputs of reduce^1 and reduce^2 as input.
19 output $(u; (x, S))$;

20 **function** reduce^3(**key**: u; **value**: $\{(x_1, S_1), [(x_2, S_2)]?\}$, where $x_1, x_2 \in \{\therefore, \boxtimes, \times\}$)
21 Create compressed node $\mathcal{S}(u) = S_1$;
22 **if** There are two values in the value list **then**
23 **if** $x_1 \neq \boxtimes$ **then** $\mathcal{S}(u).clique \leftarrow$ false;
24 **else** $\mathcal{S}(u).clique \leftarrow$ true;
25 Output $(u; \mathcal{S}(u))$;
26 **else if** $x_1 = \therefore$ **then**
27 $\mathcal{S}(u).clique \leftarrow$ false;
28 Output $(u; \mathcal{S}(u))$;

Table 1 The symbols "$\boxtimes$", "$\therefore$" and "$\times$" and their descriptions.

Symbols	Description
$\boxtimes$	A clique compressed node
$\therefore$	An independent compressed node
$\times$	Cannot determine in the context

The algorithm processes three rounds. In the first round, we aggregate the nodes that have the same closured neighbors by making $\mathcal{N}[u]$ for each $u \in V(G)$ as map^1's output key [1]. If there are more than one node gathered in the reduce^1 function ($|S| > 1$), we output the set of nodes S with u_{s_1} as the key and the symbol "$\boxtimes$" (line 10), indicating that S must form a clique compressed node with u_{s_1} as its representative node (the minimum node in the compressed node). Otherwise, we cannot determine whether the compressed node of u_{s_1} is independent or trivial at current stage, we associate the output with a "$\times$" (line 11). In the second round, the algorithm is more or less the same, but we aggregate the nodes via the neighbors of each node. Those nodes gathered by reduce^2, if more than one, must form an independent compressed node, and we associate the output with a "$\therefore$" (line 16), otherwise, a "$\times$" is attached similar to the first round (line 17). To summarize, the outputs of a node u in reduce^1 and reduce^2, denoted as out$^1(u)$ and out$^2(u)$, are related to the kinds of the compressed node of u. We have:

Proposition 1 *Given $u \in V(G)$ and its compressed node $\mathcal{S}(u)$, we have*

*(i) $\mathcal{S}(u)$ is a **trivial** compressed node **if and only if***

$$\text{out}^1(u) = (u;\ (\times, \{u\})),\ \text{out}^2(u) = (u;\ (\times, \{u\})).$$

*(ii) $\mathcal{S}(u)$ is a **clique** compressed node **if and only if***

$$\text{out}^1(u) = \begin{cases} (u;\ (\boxtimes, \mathcal{S}(u))),\ u = r_{\mathcal{S}(u)} \\ \emptyset,\ \textit{otherwise} \end{cases},$$

$$\text{out}^2(u) = (u;\ (\times, \{u\})).$$

*(iii) $\mathcal{S}(u)$ is an **independent** compressed node **if and only if***

$$\text{out}^1(u) = (u;\ (\times, \{u\})),$$

$$\text{out}^2(u) = \begin{cases} (u;\ (\therefore, \mathcal{S}(u))),\ u = r_{\mathcal{S}(u)} \\ \emptyset,\ \textit{otherwise} \end{cases}$$

Based on Proposition 1, we generate the compressed nodes in the third round. The map^3 function directly outputs $out^1(u)$ and $out^2(u)$ for each $u \in V(G)$. Given different kinds of $\mathcal{S}(u)$, we will expect one or two values in reduce^3. Note that we use $[X]$? to indicate that X may not present in the value list in the reduce function. We assume that the value with "$\therefore$" and "$\boxtimes$" will appear in the value list before the one with "$\times$". If there are two values, u either belongs to a trivial compressed node, or u is the representative node of the corresponding compressed node. We will output the compressed node after properly setting $\mathcal{S}(u).clique$ (line 23-25) [2].

Next we show the correctness of Algorithm 3.

Lemma 4 *Algorithm 3 returns each compressed node once and only once.*

Example 10 Following Example 9, we trace the outputs of node u_1 in each stage to show how Algorithm 3 runs. To start, map^1 outputs $(\mathcal{N}[u_1] = \{u_1, u_2, u_3, u_4\}; u_1)$ for u_1 (line 2). In reduce^1, $S = \{u_1, u_2, u_3\}$ is gathered on the key $\mathcal{N}[u_1]$, and $(u_1; (\boxtimes, \{u_1, u_2, u_3\}))$ is output by the reducer (line 10). In the second round,

[1] Note that we apply a threshold $\sqrt{M}$ in line 2 and line 13, and we will discuss the threshold later. Let's first assume that there is no threshold.

[2] Line 26-28 deal with the node with degree larger than the threshold (line 2,13).

map^2 outputs $(\mathcal{N}(u_1) = \{u_2, u_3, u_4\}; u_1)$ (line 13). On the key $\{u_2, u_3, u_4\}$, reduce^2 receives $S = \{u_1\}$, thus the algorithm outputs $(u_1; (\times, \{u_1\}))$ (line 17). After reduce^3 receives both $(u_1; (\boxtimes, \{u_1, u_2, u_3\}))$ and $(u_1; (\times, \{u_1\}))$ for u_1 (line 20), the clique compressed node $\mathcal{S}_1 = \mathcal{S}(u_1) = \{u_1, u_2, u_3\}$ is constructed (line 22-25). The procedures for the other nodes are similar, and we finally compute the five compressed nodes shown in Example 9.

Handling large-degree node. The node u with $d(u) \geq \sqrt{M}$ will be directly assigned to a trivial compressed node in Algorithm 3 (line 26-28). We apply this threshold to avoid a very large key that can trigger overwhelming sort cost. We show that it is highly impossible for such a node to have another equivalent node using the power-law random graph model (refer Section 6). We assume that u connects a set of nodes that have the same degree $\widehat{d}$. It is clear that $\widehat{d} < 2M/d(u)$. We now calculate the the probability pr that there exists another node u' connecting to all u's neighbors in a power-law random graph as

$$pr < (\frac{\widehat{d}\, d(u)}{2M})^{d(u)}.$$

It is clear that $\frac{\widehat{d}\, d(u)}{2M} < 1$ in a power-law random graph. We hence have $pr \approx 0$ when $d(u)$ is large enough (e.g. $d(u) > \sqrt{M}$ for a large data graph). We further show in the experiment that we can almost obtain the compressed graph by using such a threshold in all datasets.

Algorithm 4: ComprEdgeBind(data graph G, The compressed node set $V(G^*)$)

Input : G, The data graph;
$V(G^*)$, The compressed nodes, stored as $(r_{\mathcal{S}}; \mathcal{S})$ for each $\mathcal{S} \in V(G^*)$.
Output : G^*, The compressed graph.

1 **function** map_1^1(**key**: u; **value**: $\mathcal{N}(u)$)
2 output $(u; \mathcal{N}(u))$;
3 **function** map_2^1(**key**: $r_{\mathcal{S}}$; **value**: $\mathcal{S}$)
4 output $(r_{\mathcal{S}}; \mathcal{S})$;
5 **function** reduce^1(**key**: u; **value**: $\{[\mathcal{S}(u)?], \mathcal{N}(u)\}$)
6 **if** $\mathcal{S}(u)$ exists in the value list **then** output $(\mathcal{S}(u); \mathcal{N}^o(\mathcal{S}(u)) = \mathcal{N}(u) \setminus \mathcal{S}(u))$;
7 **function** map_1^2(**key**: $\mathcal{S}$; **value**: $\mathcal{N}^o(\mathcal{S})$)
8 **for each** $u' \in \mathcal{N}^o(\mathcal{S})$ **do** output $(u'; (\rightarrow, \mathcal{S}))$;
9 **function** map_2^2(**key**: $r_{\mathcal{S}}$; **value**: $\mathcal{S}$)
10 output $(r_{\mathcal{S}}; (\in, \mathcal{S}))$;
11 **function** reduce^2(**key**: u; **value**: $\{[(\in, \mathcal{S}(u))?], [(\rightarrow, \mathcal{S}(u'))?]\}$)
12 **if** both $(\in, \mathcal{S}(u))$ and $(\rightarrow, \mathcal{S}(u'))$ exist in the value list **then**
13 **if** $\mathcal{S}(u) \prec^* \mathcal{S}(u')$ **then** output $(\mathcal{S}(u); \mathcal{S}(u'))$;

Compressed-Edge Binding. Given the compressed nodes, we can construct the compressed graph by binding the compressed edges. The procedure is shown in Algorithm 4. We define the **original neighbors** of a compressed node $\mathcal{S}(u)$ as

$$\mathcal{N}^o(\mathcal{S}(u)) = \mathcal{N}(u) \setminus \mathcal{S}(u).$$

We say a data node u "connects" a compressed node $\mathcal{S}(u')$ if $\mathcal{S}(u) \neq \mathcal{S}(u')$ and $(u, u') \in E(G)$. The original neighbors of $\mathcal{S}(u)$ contains all data nodes that "connect" $\mathcal{S}(u)$. For all $u' \in \mathcal{N}^o(\mathcal{S}(u))$, it is obvious that $(\mathcal{S}(u), \mathcal{S}(u')) \in E(G^*)$. In Algorithm 4, we use the symbol "$\rightarrow$" (resp. "$\in$") to indicate that a data node connects (resp. belongs to) a compressed node. There are two rounds of executions in the algorithm. In the first round, the map function processes two inputs, namely $(u; \mathcal{N}(u))$ for all $u \in V(G)$ (line 1), and $(r_{\mathcal{S}}; \mathcal{S})$ for all $\mathcal{S} \in V(G^*)$ (line 3). The reduce^1 function then computes the original neighbors for each compressed node (line 6). In the second round, the map_1^2 function takes $(\mathcal{S}, \mathcal{N}^o(\mathcal{S}))$ for all $\mathcal{S} \in V(G^*)$ as input, and outputs $(u'; (\rightarrow, \mathcal{S}))$ for each $u' \in \mathcal{N}^o(\mathcal{S})$, indicating u' "connects" $\mathcal{S}$ (line 8). In addition, map_2^2 reads $(r_{\mathcal{S}}, \mathcal{S})$ for all $\mathcal{S} \in V(G^*)$ and outputs $(r_{\mathcal{S}}; (\in, \mathcal{S}))$ (line 10).

When the above two key-value pairs associated with the same u, namely $(u; (\rightarrow, \mathcal{S}(u')))$ and $(u; (\in, \mathcal{S}(u')))$, arrive in reduce^2, we can determine that $(\mathcal{S}(u), \mathcal{S}(u')) \in E(G^*)$ (line 13).

Lemma 5 *Algorithm 4 returns all compressed edges.*

Example 11 We have generated the compressed nodes in Example 10, we use $(\mathcal{S}_1, \mathcal{S}_4)$ as an example to show how to bind the compressed edges via Algorithm 4. Note that all other compressed edges are handled in a similar way. In the first round, map_1^1 and map_2^1 output $(u_4; \mathcal{N}(u_4) = \{u_1, u_2, u_3, u_8\})$ and $(u_4; \mathcal{S}_4 = \{u_4\})$ on the key u_4, respectively. The reduce^1 immediately computes $\mathcal{N}^o(\mathcal{S}_4) = \mathcal{N}(u_4) \setminus \mathcal{S}_4 = \{u_1, u_2, u_3, u_8\}$. In the second round, on the one hand, map_1^2 emits $(u_1; (\rightarrow, \mathcal{S}_4))$ indicating that u_1 connects $\mathcal{S}_4$; on the other hand, map_2^2 outputs $(u_1; (\in, \mathcal{S}_1))$. On the key u_1, reduce^2 discovers the compressed edge $(\mathcal{S}_1, \mathcal{S}_4)$.

Complexities. Based on Lemma 4 and Lemma 5, we have correctly built the compressed graph. Next we show that the communication cost of constructing the compressed graph is linear to the size of the data graph.

Lemma 6 *Given the data graph G, the communication cost of constructing the compressed graph of G is $O(M + N)$, where $M = |E(G)|$, and $N = |V(G)|$.*

8.2 Querying the compressed graph

We follow the join framework in Algorithm 1 to process subgraph enumeration on the compressed graph.

Considering that a compressed node can now relate to multiple data nodes, we need to accordingly redefine the matches so that we can adapt Algorithm 1 to query the compressed graph. We first introduce the *Compressed Match*.

Definition 18 (**Compressed Match**) Given a pattern graph P and a compressed graph G^*, a compressed match f^* is a mapping from $V(P)$ to $V(G^*)$ such that the following three conditions hold:

- *(Structure Preservation)* For any edge $(v_i, v_j) \in E(P)$, either $(f^*(v_i), f^*(v_j)) \in E(G^*)$ if $f^*(v_i) \neq f^*(v_j)$, or $f^*(v_i).clique = \text{true}$ if $f^*(v_i) = f^*(v_j)$.
- *(Size Limitation)* For $v_{i_1}, v_{i_2}, \ldots, v_{i_k} \in V(P)$, if $f^*(v_{i_1}) = f^*(v_{i_2}) \cdots = f^*(v_{i_k})$, then $k \leq |f^*(v_{i_1})|$.
- *(Order Preservation)* For any pair of nodes $v_i \in V(P)$ and $v_j \in V(P)$, if $v_i < v_j$ and $f^*(v_i) \neq f^*(v_j)$, then $f^*(v_i) \prec^* f^*(v_j)$.

We use $f^* = (\mathcal{S}_{k_1}, \mathcal{S}_{k_2}, \ldots, \mathcal{S}_{k_n})$ to denote the match f^*, i.e., $f^*(v_i) = \mathcal{S}_{k_i}$ for any $1 \leq i \leq n$.

It is worth noting that a compressed node $\mathcal{S}$ can now be matched up to $|\mathcal{S}|$ pattern nodes by a compressed match. Consider a pattern graph $P = (v_1, v_2, \ldots, v_k)$, a data graph G and its compressed graph G^*. We show how f and f^* can be converted to each other as follows:

- $f \rightarrow f^*$: Given a match of P in G, $f = (u_1, u_2, \ldots, u_k)$, and a bijective mapping σ where $\sigma(u) = \mathcal{S}(u)$, we obtain a compressed match of P in G^* as: $f^* = f \circ \sigma = (\mathcal{S}(u_1), \mathcal{S}(u_2), \ldots, \mathcal{S}(u_k))$.
- $f^* \rightarrow f$: Given a compressed match of P in G^*, $f^* = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k)$, and a bijective mapping σ' where $\sigma'(\mathcal{S}_i) = u$ under the conditions that: (1) $u \in \mathcal{S}_i$, (2) $\sigma'(\mathcal{S}_j) \neq u$, for any $j \neq i$, and (3) $\sigma'(\mathcal{S}_i) \prec' \sigma'(\mathcal{S}_j)$ if $v_i < v_j$, or $\sigma'(\mathcal{S}_j) \prec' \sigma'(\mathcal{S}_i)$ otherwise, we obtain a match of P in G as: $f = f^* \circ \sigma'$.

When dealing with $f^* \rightarrow f$, we replace each compressed node with one data node that satisfies all the above three conditions. The first condition says we only replace each compressed node by one of the data nodes inside it. The second condition indicates that we can only replace the compressed node by a data node that has never been used. The third condition guarantees the *Order-Preservation* constraint for a match. Note that we use the revised total order $\prec'$ (Definiton 17). Clearly, a compressed match can be mapped to multiple matches.

Example 12 Consider the square pattern graph given in Fig. 1. There are four matches of the square in the data graph presented in Fig. 6. They are: $f_1 = (u_1, u_2, u_3, u_4)$, $f_2 = (u_1, u_3, u_2, u_4)$, $f_3 = (u_1, u_2, u_4, u_3)$ and $f_4 = (u_5, u_6, u_7, u_8)$. Meanwhile, we find three compressed matches. They are: $f_1^* = (\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4)$, $f_2^* = (\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4, \mathcal{S}_1)$ and $f_3^* = (\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_5, \mathcal{S}_8)$. Among them, f_1^* compresses f_1 and f_2, f_2^* relates to f_3, and f_3^* relates to f_4.

Computing the compressed matches. As long as the compressed matches are given, it is trivial to recover the original matches, following the three conditions discussed in $f^* \rightarrow f$. We hence focus on the algorithm - SubgEnumCompr - that computes the compressed matches of P in G^*.

SubgEnumCompr follows the join framework in Algorithm 1. Recall that in Algorithm 1, with the pattern graph decomposing into $\mathcal{D}(P) = \{p_0, p_1, \ldots, p_t\}$, we enumerate the subgraph using t rounds of MapReduce. In the i^{th} round, the following join is processed:

$$R(P_i) = R(P_{i-1}) \bowtie R(p_i)$$

To process the above join, the i^{th} round MapReduce routine in Algorithm 1 will (1) compute the join attributes as $V(P_{i-1}) \cap V(p_i)$; (2) read the partial matches $R(P_{i-1})$ (computed in previous round) and map each of them according to the join key; (3) read $(u; \mathcal{N}(u))$ for each $u \in V(G)$, use them to compute $R(p_i)$, and map $R(p_i)$ to the corresponding join key; (4) process the join by filtering the results of $R(P_{i-1}) \times R(p_i)$ according to the *Conflict-Freedom* and *Order-Preservation* constraints of Definiton 11.

Denote $R^*(P)$ as the set of compressed matches. Given the same pattern decomposition, the SubgEnumCompr iteratively processes the following join using MapReduce:

$$R^*(P_i) = R^*(P_{i-1}) \bowtie R^*(p_i).$$

In order to do so, SubgEnumCompr follows the above four steps as Algorithm 1, but handles the compressed matches instead. Specifically, step (1) remains the same. In step(2) SubgEnumCompr processes $R^*(P_{i-1})$, while in step (3), it generates the compressed matches $R^*(p_i)$ of TwinTwig p_i. Finally, in step (4), SubgEnumCompr filters the results based on the *Size-Limitation* and *Order-Preservation* constraints in Definiton 18.

We first discuss how SubgEnumCompr computes $R^*(p)$ in step (3) for a TwinTwig p on the compressed graph. For ease of presentation, we assume that p is a two-edge TwinTwig (the one-edge case can be done analogously). Let $p = ((v_0, v_1), (v_0, v_2))$. Denote $R^*_{\mathcal{S}}(p) = \{f^* \mid f^*(v_0) = \mathcal{S}\}$ as the $\mathcal{S}$-specific compressed matches for some $\mathcal{S} \in V(G^*)$. Suppose G^* is stored in the form of $(\mathcal{S}; \mathcal{N}^*(\mathcal{S}))$ for each $\mathcal{S} \in V(G^*)$. SubgEnumCompr then computes $R^*_{\mathcal{S}}(p)$ on each $(\mathcal{S}; \mathcal{N}^*(\mathcal{S}))$ independently, using the algorithm in Algorithm 5.

Corollary 2 *Given a* TwinTwig p *and any compressed node* $\mathcal{S} \in V(G^*)$*, Algorithm 5 correctly computes* $R^*_{\mathcal{S}}(p)$*. Further,* $R^*(p) = \bigcup_{\mathcal{S} \in V(G^*)} R^*_{\mathcal{S}}(p)$.

Algorithm 5: ComprMatch ($(\mathcal{S}; \mathcal{N}^*(\mathcal{S}))$, p)

Input : $(\mathcal{S}; \mathcal{N}^*(\mathcal{S}))$: The compressed neighbors of $\mathcal{S}$, where $\mathcal{N}^*(\mathcal{S}) = \{\mathcal{S}_{i_1}, \mathcal{S}_{i_2}, \ldots, \mathcal{S}_{i_t}\}$, $p = ((v_0, v_1), (v_0, v_2))$: A two edge TwinTwig.
Output : $R^*_{\mathcal{S}}(p)$: The $\mathcal{S}$-specific compressed matches of p.

```
R_S(p) ← ∅
if |S| ≥ 3 and S.clique = true then
    R_S(p) = R_S(p) ∪ { (S, S, S) };
for j ∈ {1, 2, ..., t} do
    if |S| ≥ 2 and S.clique = true then
        R_S(p) = R_S(p) ∪ { (S, S, S_ij), (S, S_ij, S) };
    if |S_ij| ≥ 2 then R_S(p) = R_S(p) ∪ { (S, S_ij, S_ij) };
    for k ∈ {j + 1, ..., t} do
        R_S(p) = R_S(p) ∪ { (S, S_ij, S_ik), (S, S_ik, S_ij) };
return R_S(p);
```

We then formally show that SubgEnumCompr is correct by iteratively processing the join of compressed matches.

Lemma 7 *Given the pattern graph P, and the compressed graph G^*,* SubgEnumCompr *correctly computes all compressed matches.*

Example 13 Consider the square pattern in Fig. 1. The partial order is $v_1 < v_2 < v_4$ and $v_1 < v_3$. We show how to process the SubgEnumCompr algorithm on the compressed graph in Fig. 6. The square is partitioned into $p_0 = \{(v_1, v_2), (v_1, v_4)\}$ and $p_1 = \{(v_3, v_2), (v_3, v_4)\}$. Suppose we match the compressed node to $V(p_0)$ in the order (v_1, v_2, v_4), and $V(p_1)$ in the order (v_3, v_2, v_4). According to Algorithm 5, we find three compressed matches of p_0 in the compressed graph, namely $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1)$, $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4)$ and $(\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_8)$. And we find eight compressed matches of p_1. They are $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1)$, $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4)$, $(\mathcal{S}_4, \mathcal{S}_1, \mathcal{S}_1)$, $(\mathcal{S}_4, \mathcal{S}_1, \mathcal{S}_8)$, $(\mathcal{S}_8, \mathcal{S}_4, \mathcal{S}_5)$, $(\mathcal{S}_8, \mathcal{S}_5, \mathcal{S}_5)$, $(\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_8)$ and $(\mathcal{S}_6, \mathcal{S}_5, \mathcal{S}_5)$. To explain how the join is processed, we discuss the following three join keys:

- $(\mathcal{S}_1, \mathcal{S}_1)$: The reducer processes one partial result of p_0 - $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1)$, and two partial results of p_1 - $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1)$ and $(\mathcal{S}_4, \mathcal{S}_1, \mathcal{S}_1)$. They are joined to produce the compressed matches $f_1^* = (\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1)$ and $f_2^* = (\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4, \mathcal{S}_1)$. Among them, f_1^* is not a valid compressed match as it violates the *Size-Limitation* constraint.
- $(\mathcal{S}_1, \mathcal{S}_4)$: The reducer processes one partial result of p_0 - $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4)$, and one partial result of p_1 - $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4)$. They are joined to produce the compressed match $(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4)$.
- $(\mathcal{S}_6, \mathcal{S}_8)$: The reducer processes one partial result of p_0 - $(\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_8)$, and one partial result of p_1 - $(\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_8)$. They are joined to produce the compressed match $(\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_5, \mathcal{S}_8)$.

After we obtain the compressed matches, we resolve them to the original matches, as shown in Table 2.

Table 2 Resolve compressed matches to the original matches.

compressed match	Original Match
$(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4, \mathcal{S}_1)$	(u_1, u_2, u_4, u_3)
$(\mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_1, \mathcal{S}_4)$	(u_1, u_2, u_3, u_4), (u_1, u_3, u_2, u_4)
$(\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_5, \mathcal{S}_8)$	(u_5, u_6, u_7, u_8)

9 Performance Studies

In this section, we show our experimental results. We deployed a cluster of up to 15 computing nodes including one master node and 14 slave nodes and we used 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 allocated a JVM heap space of 1024MB for each mapper and 2048MB for each reducer, and we allowed at most 3 mappers and 3 reducers running concurrently in each machine. The block size in HDFS was set to be 128MB, the data replication factor of HDFS was set to be 3, and the I/O sort size was set to be 512MB.

Datasets. We used five real-world data graphs (see Table 3) for testing. Among them, *sk*, *lj*, *orkut*, and *fs* were downloaded from SNAP (http://snap.stanford.edu), *yt* was downloaded from KONECT (http://konect.uni-koblenz.de), and *uk*, *indo* and *arabic* were downloaded from WEB (http://law.di.unimi.it). The "r_v" and "r_e" columns in Table 3 represent the node-compression ratio of and edge-compression ratio, and are computed as $r_v = \frac{|V_h|}{|V|}$ and $r_e = \frac{|E_h|}{|E|}$, respectively. The "time / s (MR)" and "time / s (Ren)" columns write the processing time (in second) of constructing compressed graph using MapReduce with 4 computing nodes and the centralized algorithm given by [31]. Clearly, our MapReduce implementation is far more efficient than the centralized algorithm.

Algorithms. We implemented and compared seven algorithms:

- Edge: EdgeJoin (Section 4) with early filtering (Section 7.3).
- Mul: MultiwayJoin (Section 4).
- Star: StarJoin (Section 4) with early filtering (Section 7.3).
- TTBS: TwinTwigJoin (Section 5) without optimization.
- TTOA: TTBS + order-aware cost reduction (Section 7.1).
- TTLB: TTOA + workload skew reduction (Section 7.2).
- TT: TTLB + early filtering (Section 7.3).

All algorithms were implemented using Hadoop (version 1.2.1) with Java 1.6. Note that the early filtering strategy (Section 7.3) was also applied in Edge and

Table 3 Datasets used in the experiments.

dataset	name	$N = \|V\|$	$M = \|E\|$	$r_v = \frac{\|V_h\|}{\|V\|}$ (%)	$r_e = \frac{\|E_h\|}{\|E\|}$ (%)	time/s (MR)	time/s (Ren)
as-skitter	*sk*	1,696,415	11,095,298	74.87	90.98	145	2048
youtube	*yt*	3,223,589	12,223,774	45.48	79.47	165	467
live-journal	*lj*	4,847,571	42,851,237	77.37	95.16	179	1332
com-orkut	*orkut*	3,072,441	117,185,083	97.73	99.94	222	7995
indochina-2004	*indo*	7,414,758	150,984,819	**50.28**	**39.44**	124	INF
uk-2002	*uk*	18,520,486	261,787,258	**50.82**	**39.96**	264	INF
friendster	*fs*	65,608,366	1,806,067,135	65.31	99.66	2138	INF

Star, and all the optimization strategies introduced in [1] were 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 seven queries denoted by q_1 to q_7 are illustrated in Fig. 7 with edge number varying from 3 to 15 and node number varying from 3 to 6. We show the node order for symmetry breaking under each query graph. Here, we have $n \leq 5$ for most queries for fair comparison, because when n is larger than 5, except for TT, all other algorithms have very poor performance, which can be seen from the "vary-query" test for q_6. In this experiment, we only consider queries whose nodes have degree at least 2 (the "closed" queries). Non-closed queries like paths and stars often involve too many results, which can hardly be useful. For $n = 4$, we have considered all closed queries (q_1 - q_4) with edge number varying from 4 to 6 to test the influence of edge number to the performance of different algorithms.

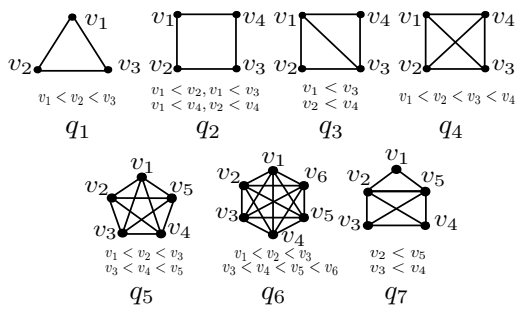
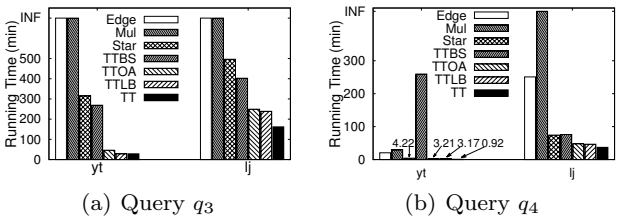


Fig. 7 Queries used in the experiment.

Exp-1: Vary Algorithms. In this experiment, we evaluated 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. 8. We also list the size of the output (see Table 4) generated by mappers and reducers in each round when we processed 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 none of the algorithms can terminate in the time limit without early filtering, since *yt* contains a lot of high-degree nodes. Thus we applied 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. 8, 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, rendering 70GB memory consumption. However, we only configured 60GB memory for all reducers in the cluster (2GB per reducer for 30 reducers). Therefore, Mul runs out of memory.



(a) Query q_3 (b) Query q_4

Fig. 8 The results of Exp-1: Vary Algorithms.

Table 4 The number of (intermediate) results for processing q_4 on *lj* (in billions).

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

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 Table 4, 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 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.

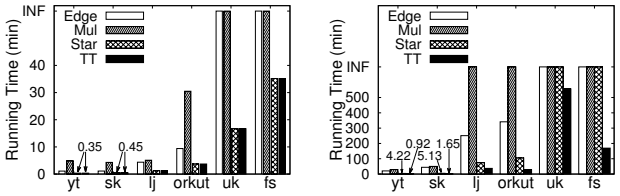
TTOA performs better than TTBS in all cases, which verifies the effectiveness of the order-aware cost reduction strategy, and TTLB outperforms TTOA in all cases, which is consistent with the analysis in Section 7.2. TT consistently outperforms all other algorithms for all test cases. Comparing TT to TTLB, we observe from Table 4 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. 8. Therefore, we focus on comparing Edge, Star, and Mul with our algorithm TT.

Exp-2: Vary Datasets. In this experiment, we tested the algorithms on all the five datasets shown in Table 3 and show our results for query q_1 and q_4 for algorithms Edge, Mul, Star, and TT.

Fig. 9(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. 9(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, the results are expected. Another interesting observation is 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 reference each other. On the contrary, *fs* is a social network with fewer large cliques than a web graph.

Exp-3: Vary Queries. We evaluated all queries q_1 to q_7 in Fig. 7. The results are illustrated in Fig. 10(a) to



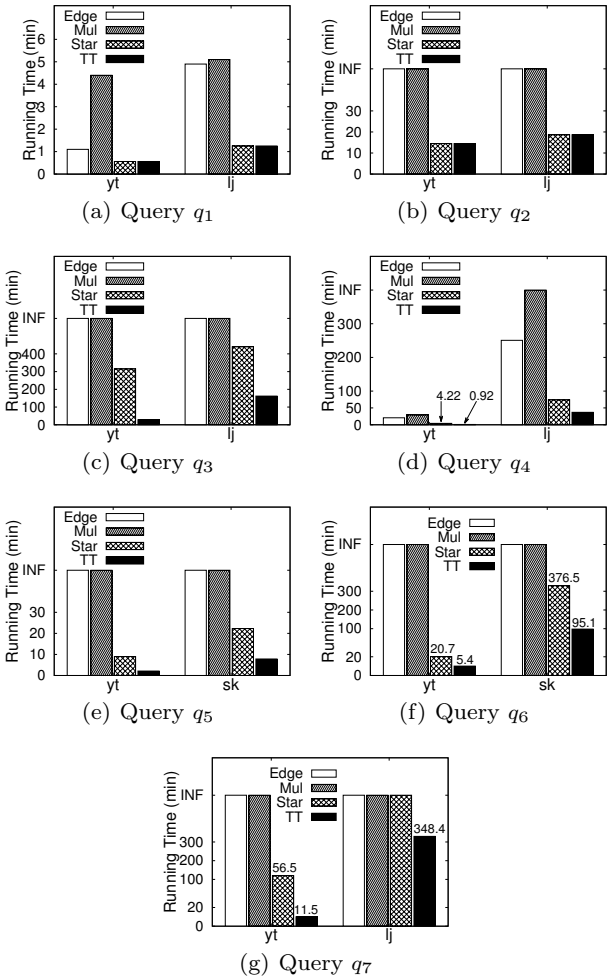
(a) Query q_1 (b) Query q_4

Fig. 9 The results of Exp-2: Vary Datasets

Fig. 10(g) 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 costly it is to evaluate the query. TT performs the best in all test cases. Note that all the tests are conducted on *yt* and *lj* except for q_5 and q_6, which is conducted on *yt* and *sk*. The reason is that, the number of results of q_5 and q_6 on *lj* is over 400 billion, which surpasses the processing ability of our current cluster. However, we can scale to handle this case by deploying more slave nodes. It is easy to find that all algorithms except TT have very poor performance while handling q_6. Edge and Mul do not response in time for both datasets. Star runs for over six hours on *sk*, a dataset of moderate size. As for q_7, a relatively complicated query, TwinTwigJoin significantly outperforms all competitors, especially on the larger dataset *lj*, where all algorithms except TwinTwigJoin cannot finish in time.

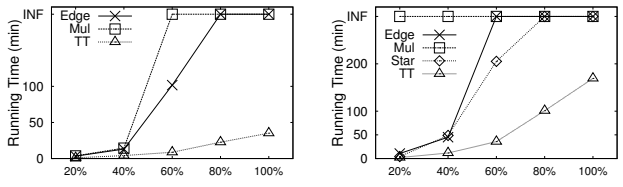
Exp-4: Vary Graph Size. We extracted subgraphs of 20%, 40%, 60%, 80%, and 100% nodes from the original graph of *fs*, and tested the algorithms using queries q_1 and q_4. The results are shown in Fig. 11(a) and Fig. 11(b) respectively. We omit the curve of Star in Fig. 11(a) since Star is the same as 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 fixed 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 tested the algorithms using queries q_1 and q_4. The results are shown in Fig. 12(a) and Fig. 12(b) respectively. We omit the curve of Star in Fig. 12(a) since Star is the same as TT when evaluating q_1. Edge and Mul fail at the very beginning. In Fig. 12(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.



(a) Query q_1 (b) Query q_2 (c) Query q_3 (d) Query q_4 (e) Query q_5 (f) Query q_6 (g) Query q_7

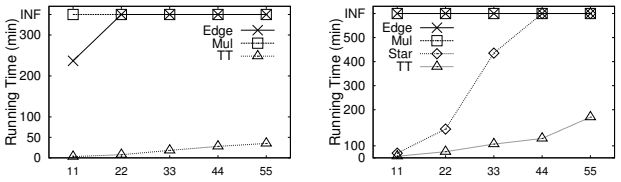
Fig. 10 The results of Exp-3: Vary Queries



(a) Query q_1 on *fs* (b) Query q_4 on *fs*

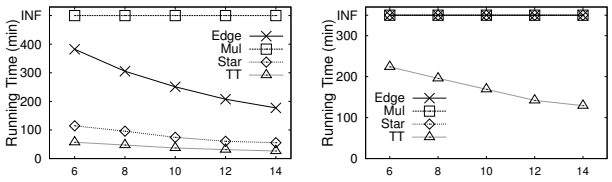
Fig. 11 The results of Exp-4: Vary Graph Size

Exp-6: Vary Slave Nodes. In this experiment, we varied the number of slave nodes from 6 to 14, and evaluated our algorithms on the *lj* and *fs* dataset using query q_4. The testing results are shown in Fig. 14(a) and Fig. 14(b) respectively. As shown in Fig. 14(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. 14(b), TT is the only algorithm that can compute the 4-clique on *fs* even when 14 slave nodes are deployed. We also performed the tests using other queries when varying slave nodes. The curves are similar to those in Fig. 12 thus are omitted due to lack of space.



(a) Query q_1 on *fs* (b) Query q_4 on *fs*

Fig. 12 The results of Exp-5: Vary Average Degree



(a) Query q_4 on *lj* (b) Query q_4 on *fs*

Fig. 13 The results of Exp-6: Vary Slave Nodes

Table 5 The ratio of intermediate results that contain only small-degree nodes (α).

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

Exp-7: Small-Degree Assumption. In this experiment, we show that the small-degree assumption A_4 (refer Section 6) is useful in practice. We call a node u with $d(u) > \sqrt{N}$ a *high-degree node*. For a data graph G, we create G^* by iteratively removing some edges of the high-degree nodes randomly until every node 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 5, 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.

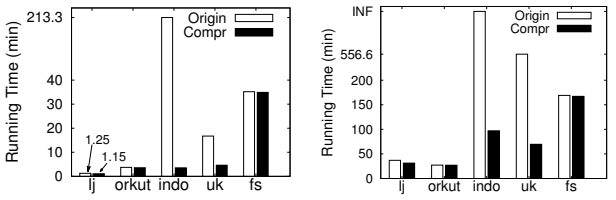
Exp-8: Compressed Graph. In Table 3, we observe a more notable compression ratio of the compressed graph built from *uk*, *indo* than the other graphs. The reason is, to our best speculation, these two graphs are web graphs, and the web pages from the same domain often reference each other, which tends to forming large

Table 6 Varying the degree threshold that makes us directly assign the node into a trivial compressed node.

datasets / threshold	$M^{0.25}$(%)		$M^{0.33}$(%)		$M^{0.5}$(%)		$M^{0.75}$(%)	
	$\varphi_v^{0.25}$	$\varphi_e^{0.25}$	$\varphi_v^{0.33}$	$\varphi_e^{0.33}$	$\varphi_v^{0.5}$	$\varphi_e^{0.5}$	$\varphi_v^{0.75}$	$\varphi_e^{0.75}$
lj	99.9253	99.1485	99.9989	99.9661	100	100	100	100
orkut	99.9999	99.9997	100	100	100	100	100	100
indo	98.1274	53.7771	99.3620	56.6513	99.9995	99.4709	100	100
uk	98.4025	70.3885	99.5810	78.9369	99.9999	99.9940	100	100

cliques. We have found large compressed nodes in the form of cliques in *uk* and *indo* of the size 943 and 6823, respectively.

Recall that we will directly assign a node u with $d(u) > \sqrt{M}$ to a trivial compressed node (i.e. $\mathcal{S}(u) = \{u\}$). We varied the threshold as $M^{0.25}$, $M^{0.33}$, $M^{0.5}$, $M^{0.75}$, M, and constructed the compressed graph accordingly in order to verify that $\sqrt{M}$ is a reasonable threshold in practice. Note that when the threshold is equal to M, we obtain the exact compressed graph. We use φ_v^i (φ_e^i) to represent the ratio of the number of the exact compressed nodes (edges) over the number of compressed nodes (edges) when the threshold is M^i (for $i \in \{0.25, 0.33, 0.5, 0.75, 1\}$). We list the experimental results for the datasets *lj*, *orkut*, *indo* and *uk* in Table 6, which cover the cases of two non-web graphs and two web graphs. We omit the other datasets as they render similar results. Clearly, $\varphi_v^1 = 100\%$ and $\varphi_e^1 = 100\%$, hence they are not presented. As we can see, when we set the threshold as $M^{0.5}$, we can obtain a compressed graph covering almost 100% compressed nodes and 100% compressed edges in all the cases.



(a) Query q_1 (b) Query q_4

Fig. 14 The results of Exp-8: Subgraph enumeration on the Compressed Graph over the original graphs.

We next compared the running time of enumerating q_1 and q_4 by using TT on both the original data graphs and the compressed graphs of *lj*, *orkut*, *indo*, *uk* and *fs*. The results are shown in Fig. 14. The performances are all improved when dealing with the compressed graphs compared to the original graphs. The improvement is especially remarkable for *uk* and *indo*. As these two graphs are web graphs, some large cliques inside them are potentially aggregated as compressed nodes. We also show the size of the output data in Table 7 for the datasets *lj*, *uk* and their compressed graphs *lj-h* and *uk-h*, while enumerating q_1 and q_4. We obtain a reduction of the output data as expected due to the compression of the compressed graph. *lj-h* produces 13% less data in the enumeration of q_1 and q_4, while *uk-h*, given a higher compression ratio as presented in Table 3, produces 74.4% and 88.80% less data in the enumeration of q_1 and q_4, respectively.

Table 7 Comparison of the size of the output data (in billions) while enumerating q_1 and q_4 on the original and compressed graph.

queries	m/r	*lj*	*lj-h*	*uk*	*uk-h*
q_1	map^1	0.33	0.29	4.71	1.30
	reduce^1	0.29	0.25	4.45	1.04
	overall	0.62	0.54	9.16	2.34
q_4	map^1	0.57	0.51	8.90	2.09
	reduce^1	9.94	8.50	157.20	17.55
	map^2	10.23	8.76	161.65	18.60
	reduce^2	9.93	8.50	157.19	17.55
	overall	30.67	26.67	484.95	55.78

10 Related Work

Subgraph Matching. Subgraph matching typically studies labeled graphs. For example, Shang et al. [33] proposed an algorithm to search from nodes with infrequent labels in order to utilize the filtering power as early as possible. Node labels in the neighborhood were utilized to filter unexpected candidates in [16] and [41]. In [15], the authors observed that a good matching order can significantly improve the performance of subgraph query. Inexact subgraph matching was also studied in [19], and [12]. Lee et al. [21] provided an in-depth comparison of subgraph isomorphism algorithms. Very recently, Ren et al. [31] has boosted subgraph matching by exploiting the node relationships in the data graph. In the label-unaware context, subgraph matching is often referred as subgraph enumeration. The centralized algorithms were studied in exact and approximate settings. The exact solutions including [7] and [14] are not scalable to handle large data graphs. The approximate solutions [4, 13, 42] only estimate the count rather than locate all the subgraph instances.

Massive Subgraph Matching. Processing subgraph matching in massive graphs has always been a hot topic due to its urgent needs. Zhao et al. [42] introduced a parallel color coding method for subgraph counting. Ma et al. [24] studied inexact graph pattern matching based on graph simulation in a distributed environment. Gonzalez et al. reported an experimental result on triangle counting in PowerGraph [17]. Recently, Sun et al. [35] proposed a subgraph matching algorithm to handle labeled graphs in the Trinity memory cloud. Graphlet, a

small induced subgraph that appears frequently in the data graph, also attracts many attentions. Ahmed et al. [2] proposed a parallel algorithm to efficiently count graphlets in a large network. Rahman et al. [30] developed the GRAFT to count the graphlets in an approximation manner. We have shown that our algorithm can be adapted to handling induced subgraph.

Subgraph Matching in MapReduce. MapReduce has been utilized to solve a lot of graph-related problems, among which subgraph enumeration (matching) has attached lots of interests. Tsourakakis et al. [37] proposed an approximate triangle counting algorithm using MapReduce. Suri et al. [36] introduced a MapReduce algorithm to compute exact triangle counting. Afrati et al. [1] proposed multiway join in MapReduce to handle subgraph enumeration. Plantenga [28] introduced an edge join method in MapReduce which can be used for subgraph enumeration. Both [1] and [28] have been introduced in details in Section 4. Moreover, the major results of this paper has appeared in the conference version [20]. Another related domain is frequent subgraph mining, which aims at enumerating all subgraphs whose appearances exceed a given threshold. Researchers solved this problem using MapReduce for efficiency and scalability considerations. Lin et al. [23] proposed the first MapReduce algorithm to search frequent subgraphs. Bhuiyan et al. [5] implemented the frequent subgraph mining algorithm based on an iterative MapReduce framework.

11 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 proposed a new TwinTwigJoin algorithm based on a left-deep-join framework in MapReduce. In the Erdös-Rényi random-graph model, we showed that under reasonable assumptions, TwinTwigJoin is instance optimal in the left-deep-join framework. An A*-based solution was given to compute the optimal join plan. In order to cover real-life applications where most graphs are power-law graphs, we proved the instance optimality of TwinTwigJoin based on the power-law random-graph model. We also improved our approach using three novel optimization strategies. Ultimately, we further improved the algorithm by constructing the compressed graph regarding the equivalence relationships among the data nodes. We conducted extensive performance studies on real large graphs with up to billions of edges to demonstrate the effectiveness of our approach.

References

1. Afrati, F.N., Fotakis, D., Ullman, J.D.: Enumerating subgraph instances using map-reduce. In: Proc. of ICDE'13 (2013)
2. Ahmed, N.K., Neville, J., Rossi, R.A., Duffield, N., Willke, T.L.: Graphlet Decomposition: Framework, Algorithms, and Applications. ArXiv e-prints (2015)
3. Aiello, W., Chung, F., Lu, L.: A random graph model for massive graphs. In: Proc. of STOC '00 (2000)
4. Alon, N., Dao, P., Hajirasouliha, I., Hormozdiari, F., Sahinalp, S.C.: Biomolecular network motif counting and discovery by color coding. In: Proc. of ISMB'08 (2008)
5. Bhuiyan, M.A., Hasan, M.A.: An iterative mapreduce based frequent subgraph mining algorithm. TKDE **27**(3), 608–620 (2015)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7) (1970)
7. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1) (1985)
8. Chung, F.R.K., Lu, L., Vu, V.H.: The spectra of random graphs with given expected degrees. Internet Mathematics **1**(3) (2003)
9. Clauset, A., Shalizi, C.R., Newman, M.E.J.: Power-law distributions in empirical data. SIAM Rev. (2009)
10. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proc. of OSDI'04 (2004)
11. Erdos, P., Renyi, A.: On the evolution of random graphs. In: Publ. Math. Inst. Hungary. Acad. Sci. (1960)
12. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph pattern matching: From intractable to polynomial time. PVLDB **3**(1) (2010)
13. Gonen, M., Ron, D., Shavitt, Y.: Counting stars and other small subgraphs in sublinear time. In: Proc. of SODA'10 (2010)
14. Grochow, J.A., Kellis, M.: Network motif discovery using subgraph enumeration and symmetry-breaking. In: Proc. of RECOMB'07 (2007)
15. Han, W.S., Lee, J., Lee, J.H.: Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proc. of SIGMOD'13 (2013)
16. He, H., Singh, A.K.: Graphs-at-a-time: Query language and access methods for graph databases. In: Proc. of SIGMOD'08 (2008)
17. J.Gonzalez, Y.Low, H.Gu, D.Bickson, C.Guestrin: Powergraph:distributed graph-parallel computation on natural graphs. In: Proc. of OSDI'12 (2012)
18. Kairam, S.R., Wang, D.J., Leskovec, J.: The life and death of online groups: Predicting group growth and longevity. In: Proc. of WSDM'12 (2012)
19. Khan, A., Wu, Y., Aggarwal, C.C., Yan, X.: Nema: Fast graph search with label similarity. PVLDB **6**(3) (2013)
20. Lai, L., Qin, L., Lin, X., Chang, L.: Scalable subgraph enumeration in mapreduce. Proc. VLDB Endow. **8**(10), 974–985 (2015)
21. Lee, J., Han, W.S., Kasperovics, R., Lee, J.H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB **6**(2) (2012)
22. Leskovec, J., Singh, A., Kleinberg, J.: Patterns of influence in a recommendation network. In: Proc. of PAKDD'06 (2006)
23. Lin, W., Xiao, X., Gabriel, G.: Large-scale frequent subgraph mining in mapreduce. In: ICDE, pp. 844–855 (2014)
24. Ma, S., Cao, Y., Huai, J., Wo, T.: Distributed graph pattern matching. In: WWW (2012)
25. Milenkovic, T., Przulj, N.: Uncovering biological network function via graphlet degree signatures. Cancer Inform **6** (2008)

26. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: Simple building blocks of complex networks. Science **298**(5594) (2002)
27. N.Shervashidze, S.Vishwanathan, T.Petri, K.Mehlhorn, K.Borgwardt: Efficient graphlet kernels for large graph comparison. In: AISTATS (2009)
28. Plantenga, T.: Inexact subgraph isomorphism in mapreduce. J. Parallel Distrib. Comput. **73**(2) (2013)
29. Przulj, N.: Biological network comparison using graphlet degree distribution. Bioinformatics **23**(2) (2007)
30. Rahman, M., Bhuiyan, M.A., Hasan, M.A.: Graft: An efficient graphlet counting method for large graph analysis. TKDE **26**(10), 2466–2478 (2014)
31. Ren, X., Wang, J.: Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. Proc. VLDB Endow. **8**(5), 617–628 (2015)
32. Rücker, G., Rücker, C.: Substructure, subgraph, and walk counts as measures of the complexity of graphs and molecules. Journal of Chemical Information and Computer Sciences **41**(6) (2001)
33. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. PVLDB **1**(1) (2008)
34. Steinbrunn, M., Moerkotte, G., Kemper, A.: Optimizing join orders. Tech. rep. (1993)
35. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. PVLDB **5**(9) (2012)
36. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: Proc. of WWW'11 (2011)
37. Tsourakakis, C.E., Kang, U., Miller, G.L., Faloutsos, C.: Doulion: Counting triangles in massive graphs with a coin. In: Proc. of KDD'09 (2009)
38. Viger, F., Latapy, M.: Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In: COCOON'05, pp. 440–449 (2005)
39. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB **5**(9) (2012)
40. Watts, D., Strogatz, S.: Collective dynamics of 'small-world' networks. Nature **6684**(393) (1998)
41. Zhao, P., Han, J.: On graph query optimization in large networks. PVLDB **3**(1-2) (2010)
42. Zhao, Z., Khan, M., Kumar, V.S.A., Marathe, M.V.: Subgraph enumeration in large social contact networks using parallel color coding and streaming. In: Proc. of ICPP'10 (2010)

APPENDIX

A.1 Proofs in Section 5

Proof of Lemma 2 (Section 5.2). 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

$$|R(P_i)| = |R(P_{i-1})| \times (\frac{2M}{N^2})^{\Delta m_i} \times N^{\Delta n_i}. \quad (8)$$

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, $|R(P_i)| \le |R(P_{i-1})| \times \frac{(2M)^2}{N^3} \le |R(P_{i-2})| \times (\frac{(2M)^2}{N^3})^2 \le \ldots \le |R(p_0)| \times (\frac{(2M)^2}{N^3})^i$. □

Proof of Corollary 1 (Section 5.2). By the 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 1 holds according to Lemma 2. □

Proof of Theorem 1 (Section 5.2). 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 Lemma 1, 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

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

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

$$\mathsf{cost}_2(\mathcal{D}') = \Theta(\Sigma_{i=1}^{t'}(|R(P'_{i-1})| + |R(P'_i)|)). \quad (9)$$

Note that in $\mathcal{D}$ that is constructed based on $\mathcal{D}'$, we will gradually combine $p^i_1, p^i_2, \ldots, p^i_{t_i}$ 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

$$\begin{aligned}\mathsf{cost}^i_2(\mathcal{D}) = |R(P'_{i-1})| + |R(P'_{i-1} \cup p^i_1)| + \\ \cdots + |R(P'_{i-1} \cup p^i_1 \cup \cdots \cup p^i_{t_i-1})| + |R(P'_i)|.\end{aligned} \quad (10)$$

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 2 and Corollary 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 $\mathsf{cost}_2^i(\mathcal{D})$ is $\Theta(|R(P_i')|)$. Since each p_j^i is a non-strong TwinTwig, according to Eq. 8, when j increases, the size of the j-th term increases exponentially with a rate $\geq d > 1$, 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 therefore, Theorem 1 holds. □

Proof of Lemma 3 (Section 5.3). 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

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

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

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

A.2 Instance Optimality of TwinTwigJoin in the power-law random graph (Section 6).

To show that the instance optimality of TwinTwigJoin in power-law graphs, we prove that Theorem 1 holds in a power-law random graph model. Following the same proof structure as Theorem 1, we divide the proof into the following two parts: In part 1, we prove that $\mathsf{cost}_1(\mathcal{D}) \leq \Theta(\mathsf{cost}_1(\mathcal{D}'))$, 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. 9 and Eq. 10, 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. Below is the detailed proof.

(Part 1): Let p be a two-edge TwinTwig[3], we have

$$\begin{aligned}
&\mathsf{cost}_1(\mathcal{D}^i) = \Theta(|R(p)| \cdot t_i') \text{ and,} \\
&\mathsf{cost}_1(\{p_i'\}) = \Theta(|R(p)| \cdot \mathbb{E}[d(u)^{t_i'-2}]) \\
&\qquad \geq \Theta(|R(p)| \cdot \mathbb{E}[d(u)]^{t_i'-2}) = \Theta(|R(p)| \cdot d^{t_i'-2}).
\end{aligned}$$

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

$$\begin{aligned}
\gamma &= \mathbb{E}[\Sigma_{u \in V(G)} d(u') d(u) \rho \times d(u'') d(u) \rho] \\
&= \mathbb{E}[d(u') d(u'')] \times \rho^2 \Sigma_{i=1}^N w_i^2.
\end{aligned}$$

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

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

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:

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

As a result, we have

$$\begin{aligned}
\mathbb{E}[d(u')d(u'')] &= \Sigma_{i,j=1}^N \rho P_{i,j} w_i w_j \\
&= \rho^2 (\Sigma_{i=1}^N w_i^2 \Sigma_{j=1}^N w_j^2) = \rho^2 (\Sigma_{i=1}^N w_i^2)^2.
\end{aligned}$$

Therefore, γ can be calculated as

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

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 [38], and thus γ can be calculated via Eq. 13. The results are shown in Fig. 15, 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

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

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.

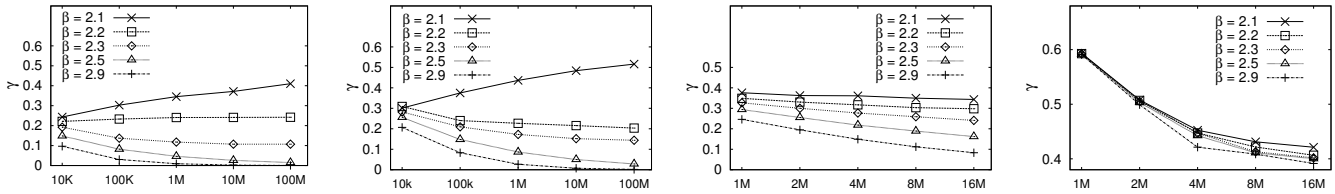
According to Part 1 and Part 2, the instance optimality of the TwinTwigJoin holds for a power-law random graph.

A.3 Proofs in Section 8

Proof of Proposition 1 (Section 8.1). (i) is apparently true, and the proof of (iii) is similar to (ii), hence we concentrate on (ii) here.

(**If**) Let S_1 be the set of nodes aggregated on $\mathcal{N}[u]$ in reduce[1] (Algorithm 3) and S_2 be the set of nodes aggregated

[3] The case is much easier when p is an edge, we hence focus on the two-edge TwinTwig in the following.



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

Fig. 15 The values of γ in different parameter combinations

on $\mathcal{N}(u)$ in reduce^2 (Algorithm 3). If $\mathcal{S}(u)$ is a clique compressed node, we show that (1) $\mathcal{S}(u) = S_1$, and (2) $S_2 = \{u\}$.

(1) On the one way, $\forall u' \in \mathcal{S}(u)$, and $u' \neq u$, we know $\mathcal{N}[u'] = \mathcal{N}[u]$, and u' must be aggregated in reduce^1 (Algorithm 3) on the key $\mathcal{N}[u]$. Thus, $u' \in S_1$, and as a result, $S_1 \subseteq \mathcal{S}(u)$. On the other way, $\forall u' \in S_1$ and $u' \neq u$, we have $\mathcal{N}[u] = \mathcal{N}[u']$, leading to $\mathcal{N}(u') \setminus \{u\} = \mathcal{N}(u) \setminus \{u'\}$. According to Definiton 13 and Definiton 14, we have $u' \in \mathcal{S}(u)$. As a result, $\mathcal{S}(u) \subseteq S_1$. Conclusively, $\mathcal{S}(u) = S_1$ holds. Note that we only output the record in line 10 (Algorithm 3) for u_{s_1}, the minimum node in S_1 (also the representative node of $\mathcal{S}(u)$). Therefore, we have out$^1(u)$ as shown in 2.

(2) It suffices to show that $\nexists u' \neq u$, such that $\mathcal{N}(u') = \mathcal{N}(u)$. We prove this by contradiction. Suppose there is such a u'. By $\mathcal{N}(u') = \mathcal{N}(u)$, we must have $u' \notin \mathcal{N}(u)$. As $\mathcal{S}(u)$ is a clique compressed node, $\exists u'' \neq u'$ and $u'' \neq u$, such that $\mathcal{N}[u''] = \mathcal{N}[u]$. We hence have $u'' \in \mathcal{N}(u) \Rightarrow u'' \in \mathcal{N}(u') \Rightarrow u' \in \mathcal{N}(u'') \Rightarrow u' \in \mathcal{N}(u)$. This draws a contradiction. As a result, there are not nodes but u itself gathered in reduce^2 (Algorithm 3), and we have out^2 as shown in 2.

(**Only If**) While $u = r_{\mathcal{S}(u)}$ and having out$^1(u) = (u; (\boxtimes, \mathcal{S}(u)))$, it is apparent that $\mathcal{S}(u)$ is a clique compressed node. Otherwise, out$^1(u) = \emptyset$. Clearly u does not belong to a trivial compressed node, as otherwise 1 is expected. Additionally, $\mathcal{S}(u)$ cannot be an independent compressed node, as $out^2(u)$ would never be associated with a "$\times$" if this is the case. Therefor, $\mathcal{S}(u)$ must be a clique compressed node. □

Proof of Lemma 4 (Section 8.1). It is clear that each trivial compressed node $\mathcal{S} = \{u\}$ will be output in reduce^3 (Algorithm 3) on the key u. Consider a non-trivial compressed node $\mathcal{S} = \{u_{s_1}, u_{s_2}, \ldots, u_{s_k}\}$. According to Proposition 1, reduce^3 (Algorithm 3) will receive two values only on the key u_{s_1}, where the compressed node $\mathcal{S}$ is generated with u_{s_1} as the representative node. Therefore, the lemma holds. □

Proof of Lemma 5 (Section 8.1). Given any compressed edge $(\mathcal{S}, \mathcal{S}') \in E(G^*)$, we show it is returned by Algorithm 4. Let $u = r_{\mathcal{S}}$. On the one hand, map$_2^2$ (Algorithm 4) outputs $(u; (\in, \mathcal{S}))$. On the other hand, we have $u \in \mathcal{N}^o(\mathcal{S}')$ when $(\mathcal{S}, \mathcal{S}') \in E(G^*)$. As a result, map$_1^2$ (Algorithm 4) involves $(u; (\rightarrow, \mathcal{S}'))$ in the output. Finally, the above two key-value pairs arrive at reduce^2 (Algorithm 4), and the corresponding compressed edge is binded. □

Proof of Lemma 6 (Section 8.1). In MapReduce, communication cost is triggered by transferring the output data of each mapper to the reducer. In Algorithm 3, map^1 and map^2 output the neighbors for each node, and the cost is $O(M)$, and map^3 outputs each node with its compressed node, and the cost is $O(N \cdot \overline{|\mathcal{S}|})$, where $\overline{|\mathcal{S}|}$ is the average size of the compressed nodes. In Algorithm 4, map$_1^1$ outputs each node with its neighbors and map$_2^1$ outputs the representative node with its compressed node. They contribute to $O(M + N \cdot \overline{|\mathcal{S}|})$ cost. As for map$_1^2$ and map$_2^2$, we can simply use $r_{\mathcal{S}(u)}$ to represent $\mathcal{S}(u)$, hence they render the same cost as the first stage. To summarize, the overall communication cost of the construction of compressed graph is $O(M+N \cdot \overline{|\mathcal{S}|})$, or simply $O(M+N)$ considering that $\overline{|\mathcal{S}|}$ is often small. □

Proof of Corollary 2 (Section 8.2). Given a match of p (u_0, u_1, u_2), such that the corresponding compressed match $(\mathcal{S}(u_0), \mathcal{S}(u_1), \mathcal{S}(u_2))$ satisfies $\mathcal{S}(u_0) = \mathcal{S}$. As a valid match of p, we must have $(u_0, u_1) \in E(G)$ and $(u_0, u_2) \in E(G)$. There are four cases for the compressed match.

- $\mathcal{S} = \mathcal{S}(u_0) = \mathcal{S}(u_1) = \mathcal{S}(u_2)$. In this case, $\mathcal{S}$ at least includes $\{u_0, u_1, u_2\}$. Further, we have $\mathcal{S}$.clique=true due to $(u_0, u_1) \in E(G)$. This compressed match is handled in line 2 in Algorithm 5.
- $\mathcal{S} = \mathcal{S}(u_1)$ or $\mathcal{S} = \mathcal{S}(u_2)$. In this case, $\mathcal{S}$ has at least two nodes and similarly $\mathcal{S}.clique$ =true. This compressed match is processed in line 4.
- $\mathcal{S} \neq \mathcal{S}(u_1) = \mathcal{S}(u_2)$. Note that $\mathcal{S}(u_1) \in \mathcal{N}^*(\mathcal{S})$, and this case is covered in line 5.
- $\mathcal{S} \neq \mathcal{S}(u_1) \neq \mathcal{S}(u_2)$. Both compressed nodes are $\mathcal{S}$'s neighbors. Algorithm 5 covers this case in line 7 by enumerating the pairs of compressed nodes in $\mathcal{S}$'s compressed neighbors.

Summarizing the above cases, Algorithm 5 returns all $R^*_{\mathcal{S}}(p)$. It is obvious that $R^*(p) = \bigcup_{\mathcal{S} \in V(G^*)} R^*_{\mathcal{S}}(p)$. This completes the proof. □

Proof of Lemma 7 (Section 8.2). Following the pattern decomposition $\mathcal{D}(P) = \{p_0, p_1, \ldots, p_t\}$, the algorithm processes t rounds. We prove this lemma by making inductions on the MapReduce rounds.

Initially, it is round 0 where P_0 is a TwinTwig. The lemma holds as SubgEnumCompr correctly computes all compressed matches of a TwinTwig according to Corollary 2.

Suppose SubgEnumCompr correctly computes all compressed matches of P_{n-1} in the $(n-1)^{th}$ round, where $1 < n \leq t$. In this n^{th} round, we know that SubgEnumCompr will process the join $R^*(P_n) = R^*(P_{n-1}) \bowtie R^*(p_n)$. Let the join attributes be $V_k = V(P_{n-1}) \cap V(p_n)$ and $V(P_n) = (V(P_{n-1}) \setminus V_k, V_k, V(p_n) \setminus V_k)$. Given a match of P_n - f - we divide it into three parts, namely $f_{n-1} = f(V(P_{n-1}) \setminus V_k)$, $f_k = f(V_k)$ and $f_n = f(V(p_n) \setminus V_k))$, where $f(V) = (f(v_1), f(v_2), \ldots)$ for all $v_j \in V$. Define a bijective mapping $\sigma : V(G) \mapsto V(G^*)$ such that $\sigma(u) = \mathcal{S}(u)$ for all $u \in V(G)$. The compressed match related to f, can hence be written as $f \circ \sigma = (f_{n-1} \circ \sigma, f_k \circ \sigma, f_n \circ \sigma)$. It is obviously that $(f_{n-1} \circ \sigma, f_k \circ \sigma) \in R^*(P_{n-1})$ and $(f_k \circ \sigma, f_n \circ \sigma) \in R^*(p_n)$. According to the induction and Corollary 2, the algorithm correctly computes all $R^*(P_{n-1})$ and $R^*(p_n)$. Therefore, $(f_{n-1} \circ \sigma, f_k \circ \sigma)$ and $(f_k \circ \sigma, f_n \circ \sigma)$ must have been computed and will be joined in this round on the key $f_k \circ \sigma$ to generate the compressed match of f. In other words, any compressed match in $R^*(P_n)$ that is related to a valid match will be correctly computed.

By induction, SubgEnumCompr correctly computes all compressed matches of P after t rounds of MapReduce. □