Distributed Subgraph Enumeration

Distributed Subgraph Enumeration

by

Longbin Lai

B.E. Shanghai Jiao Tong University, 2010

M.E. Shanghai Jiao Tong University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

IN THE SCHOOL

OF

Computer Science and Engineering

UNSW
AUSTRALIA

Wednesday 21st June, 2017

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: LAI

First name: LONGBIN Other name/s:

Abbreviation for degree as given in the University calendar: PhD

School: School of Computer Science and Engineering Faculty: Faculty of Engineering

Title: Distributed subgraph enumeration

Abstract 350 words maximum: (PLEASE TYPE)

Subgraph enumeration is a fundamental graph problem with many applications. However, existing algorithms for subgraph enumeration fall short in handling large graphs due to the computational hardness. In this work, we propose a general approach that solves subgraph enumeration in the distributed contexts, including MapReduce and Spark. The approach features a decomposition-and-join manner, in which the pattern graph is decomposed into a set of structures, called join unit. We introduce the distributed graph storage mechanism to determine what structure can be the join unit. Consequently, we obtain the results by joining the matches of all join units following a specific join structure. Based on the general approach, we first propose a star-based join framework. In the framework, we adopt a basic graph storage mechanism that only supports a star (a tree with depth 2) as the join unit, and we apply the left-deep join structure to process the join. We then show that a special star called TwinTwig - an edge or two incident edges of a node - is enough to guarantee instance optimality in the star-based join framework under reasonable assumptions, which inspires the TwinTwigJoin algorithm. We devise an A*-based algorithm to compute the optimal join plan for TwinTwigJoin. TwinTwigJoin is still not scalable to large graph because of the constraints in the left-deep join structure and that each join unit must be a star. We then explore the graph-based join framework that allows us to use more than just star as the join unit. In addition, we use the bushy join structure rather than left-deep join to guarantee the optimality of the join plan. Aware that it is storage-inefficient to use any structure as the join unit, we develop the SEED algorithm that implements an effective distributed graph storage mechanism to use star and clique (complete graph) as the join units. We then devise a dynamic-programming algorithm to compute an optimal bushy join plan. SEED frees us from the constraints in TwinTwigJoin, and greatly improves the performance of subgraph enumeration. Ultimately, we develop two data-compression techniques, namely compressed graph and clique compression, to further reduce the enormous cost while transferring and maintaining the (intermediate) results.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

21/06/2017

Signature Witness Signature Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed ……..

Date ……..

Abstract

Subgraph enumeration is a fundamental graph problem with many applications. However, existing algorithms for subgraph enumeration fall short in handling large graphs due to the computational hardness. In this work, we propose a general approach that solves subgraph enumeration in the distributed contexts, including MapReduce and Spark. The approach features a decomposition-and-join manner, in which the pattern graph is first decomposed into a set of structures, called join unit. We propose the distributed graph storage mechanism to determine what structure can be the join unit. Consequently, we obtain the results by joining the matches of all join units following a specific join structure. Based on the general approach, we first propose a star-based join framework. In the framework, we adopt a basic graph storage mechanism that only supports a star (a tree with depth 2) as the join unit, and we apply the left-deep join structure to process the join. We then show that a special star called TwinTwig- an edge or two incident edges of a node - is enough to guarantee instance optimality in the star-based join framework under reasonable assumptions, which inspires the TwinTwigJoin algorithm. We devise an A*-based algorithm to compute the optimal join plan for TwinTwigJoin. TwinTwigJoin is still not scalable to large graph because of the constraints in the left-deep join structure and that each join unit must be a star. We then explore the graph-based join framework that allows us to use more than

just star as the join unit. In addition, we use the bushy join structure rather than left-deep join to guarantee the optimality of the join plan. Aware that it is storage-inefficient to use any structure as the join unit, we develop the SEED algorithm that implements an effective distributed graph storage mechanism to use star and clique (complete graph) as the join units. We then devise a dynamic-programming algorithm to compute an optimal bushy join plan. SEED frees us from the constraints in TwinTwigJoin, and greatly improves the performance of subgraph enumeration. Ultimately, we develop two data-compression techniques, namely compressed graph and clique compression to further reduce the enormous cost while transferring and maintaining the (intermediate) results.

Publications Involved in Thesis

- Longbin Lai , Lu Qin, Xuemin Lin, Lijun Chang. Scalable Subgraph Enumeration in MapReduce, in VLDB, 2015. (Chapter 4)
- Longbin Lai , Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang. Scalable Distributed Subgraph Enumeration, in VLDB, 2017.(Chapter 3, Chapter 5 and Chapter 6)
- Longbin Lai , Lu Qin, Xuemin Lin, Lijun Chang. Scalable Subgraph Enumeration in MapReduce - A Cost-Oriented Approach, under reviewed, VLDB Journal. (Chapter 6)

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Xuemin Lin for the continuous support of my Ph.D study and related research, for his patience, motivation, immense knowledge, and financial support. His guidance helped me in all the time of research and writing of this thesis. I can always remember his motto: "A good researcher is an outstanding hunter in the dark forest. He who embraces the loneliness, desperation and bitterness would ultimately be granted what he hunts for."

A special thanks goes to my joint-advisor, Dr. Lu Qin, who is most responsible for helping me complete the writing of the academic papers and this dissertation as well as the challenging research that lies behind them. Lu has been a friend and mentor. He taught me how to write papers, and as an ACM competition winner, instructed me to be a better programmer. He always had confidence in me, even when I deeply doubted myself after a whole-year futile work in finding the research topic. He guided me all the way out with his continuous brilliant ideas and inspirations.

The work in this thesis can not be accomplished without the guidance of Dr. Lijun Chang and Dr. Ying Zhang, whose creativity, passion and hardworking not only light up my candle of hope when it is nearly extinguished by the sense of desperation, but only set up high-end examples of outstanding researchers.

I would also like to thank Dr. Wenjie Zhang, Prof. Wei Wang, Dr. Jianbin Qin and Dr. John Shepherd for annually reviewing my Ph.D process. I recognised that you had always given me so many credits in the review reports, much more than what I had fulfilled, which helped me build my confidence till finally completing the Ph.D thesis.

Let me also say "thank you" to the following people at UNSW, Australia: Mr. Simon Garrod (Technical Support) and Mr. Tianlun Bill for your consistent help in my English. Mr. Philip Rodwell (Technical Support) and Craig Howie (IT manager) for sorting everything out, even for my stand-working facility. Dr. Zengfeng Huang, Dr. Xin Cao, Dr. Muhammad Aamir Cheema, Dr. Chengyuan Zhang, Dr. Xiang Zhao and Dr. Gaoping Zhu, for sharing your brilliant ideas and experiences. Dr. Xiaoyang Wang and Dr. Shiyu Yang, for your restless assistance whenever I need. Mr. Xiang Wang, Mr. Long Yuan, Mr. Xing Feng, Mr. Jianye Yang, Miss Lu Shen, Mr. Fei Bi, Mr. Dong Wen, Mr. Fan Zhang, Mr. Haida Zhang, Miss Chen Zhang and Mr. Wei li, for sharing the happiness and bitterness with me during my Ph.D study.

I am also greatly indebted to many teachers in Shanghai Jiao Tong University: Prof. Minyi Guo, Prof. Kefei Chen, Prof. Jingyu Zhou and Prof. Wujun Li for arousing my interests in research, and referencing me for my Ph.D study. Miss Chunling Zhu, for your guidance of my life and study since fresh year in the university.

Last but not least, I thank my family: my mother, Xiubi Lai, for bringing me to this world and for educating me to be a descent man. My sister Yuru Lai and my brother Junyong Lai, for sharing their experiences of life with me, for listening to my complaints and frustrations, and for believing in me. My girlfriend, Miao Wang, for her selfless support, persistent encouragement and ceaseless company.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this article, 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 [MSOI$^+$02, ADH$^+$08] to facilitate the design of large network from biochemistry, neurobiology, ecology, and bioinformatics. It is utilized to compute the graphlet kernels for large graph comparison [SVP$^+$09, Prz07] and property generalization for biological networks [MP08]. It is considered as a key operation for the synthesis of target structures in chemistry [RR01]. It is also adopted to illustrate the evolution of social networks [KWL12] and to discover the information trend in recommendation networks [LSK06]. In addition, as a special case of subgraph enumeration, triangle enumeration is a preliminary operation in cluster coefficient calculation [WS98] and community detection [WC12].

1.1 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 [CN85, GK07] are not scalable to big graphs. Some other studies try to find approximate solutions [ADH+08, GRS10, ZKKM10] to reduce the computational cost, however, they only estimate the count of the matched subgraphs rather than locate all the subgraph instances.

Researchers hence seek distributed solutions for scalability consideration, which typically leverage big data (graph) engines such as MapReduce [DG04] and Pregel [MAB+10]. As representatives, there are two existing approaches in MapReduce, namely, EdgeJoin [Pla13] and MultiwayJoin [AFU13], and one work in Pregel called PSgL [SCC+14].

In EdgeJoin [Pla13], 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. EdgeJoin is inefficient as joining one edge in each round cannot fully make use of the structural information, which may render numerous partial results. In MultiwayJoin [AFU13], 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, MultiwayJoin usually encounters serious scalability problems by keeping almost the

whole graph in the memory of each machine when the pattern graph is complex.

PSgL is Pregel-based and processes subgraph enumeration via graph traversal opposed to join operation. The algorithm applies a breadth-first-search strategy - that is, each time it picks up an already-matched but not fully-expanded node v, and searches the matches of its neighbors in order to generate finer-grained results. PSgL is efficient while processing small dataset, benefiting from Pregel's in-memory computation (maintaining the result in memory), but it suffers from severe memory issue immediately when data graph becomes large.

1.2 Our Approach

Important as the subgraph enumeration though, there lacks an efficient and scalable solution in the literature. To close this gap, we propose a general decomposition-and-join approach that can be implemented in the general-purposed big data process engine, including MapReduce [DG04], Spark [MMJ+], Dryad [IBY+] and Myria [HTC+]. For simplicity, we will introduce all proposed algorithms using MapReduce. Given the hardness of the subgraph enumeration, we first decompose the pattern graph into a set of easier-solving structures, called *join unit*. We then introduce the distributed graph storage mechanism, which determines what structure can serve as the join unit. Consequently, we obtain the results by joining the matches of all join units following a certain join structure. The whole procedure processes in multiple rounds, and each of them handles a two-way join. In order to evaluate the proposed algorithms (and their optimality), we carefully compute the cost, taking into account the cost of transferring and maintaining the (partial) result set in each algorithm.

Based on the general approach, we first propose the star-based join framework.

In the framework, we adopt the basic graph storage mechanism, in which a star (a tree with depth 2) is the join unit. After decomposing the pattern graph into a set of disjoint stars, we join the matches of these stars in a left-deep join structure. We will show that the star-based join framework can generalize the EdgeJoin and StarJoin algorithms. However, it is sometimes inefficient to process a star due to the enormous results produced. For example, a *celebrity node* with 1,000,000 neighbors in the social network would incur $O(10^{18})$ matches of a star of three edges. One such large-degree node alone would exhaust both the computation and storage in any machine and become a huge bottleneck of the algorithm. Aware of the deficiency, we propose the TwinTwigJoin algorithm that decomposes the pattern graph into TwinTwig- a star of either one or two edges - instead of a general star. The TwinTwigJoin has several advantages. First, based on a well-defined cost model as well as a variant of Erdös Rényi random (ER) graph model [ER60], we show that TwinTwigJoin can ensure instance optimality in the star-based join framework. Second, the simple structure of a TwinTwig makes it easy to devise an A*-based algorithm to compute the optimal left-deep join plan. The algorithm runs with space and time complexities of $O(2^m)$ and $O(d_{max} \cdot m \cdot 2^m)$ respectively, where m is number of edges and d_{max} is the maximum degree in the pattern graph. Third, a lot of optimization strategies can be designed on top of TwinTwigJoin, including order-aware cost reduction, workload skew reduction, and early filtering.

TwinTwigJoin only guarantees optimality under two constraints: (1) each join unit is a star, and (2) the join structure is left-deep. These constraints hamper its practicality in several respects. First, TwinTwigJoin only mitigates but not resolves the issues in the star-based join. For example, the node of degree 1,000,000 still produces $O(10^{12})$ matches of a two-edge TwinTwig. Second, it takes TwinTwigJoin at least $\frac{m}{2}$ (m is the number of pattern edges) rounds to solve subgraph enumera-

tion, making it inefficient to handle complex pattern graph. Finally, the algorithm utilizes a left-deep join plan, which may result in a sub-optimal solution [JK84]. Last but not least, TwinTwigJoin bases the cost analysis on the ER model , which can be biased considering that most real-life graphs are power-law graphs.

Targeting the deficiencies of TwinTwigJoin, we further explore the graph-based bushy join framework so as to use more than just star, but any structure, as the join units. After noticing that it is storage-inefficient to consider any structure as join unit, we develop the SEED ((**S**ubgraph **E**num**E**ration in **D**istributed context)) algorithm that implements the star-clique-preserved (SCP) storage mechanism to support star and clique (a complete graph) as join units. With clique as an alternative, we can make a better choice other than star, where possible, and reduce the number of execution rounds. Ultimately, this leads to a huge reduction of the intermediate results. In addition, we refine the cost model in TwinTwigJoin by basing the cost analysis on the power-law random (PR) graph model [CLV03a] instead of the ER model. Considering that many real graphs are power-law graphs, the PR model offers more realistic estimation than the ER model. Finally, we develop a dynamic-programming algorithm to compute the **optimal** bushy join plan. With the same space complexity and a slightly larger time complexity $O(3^m)$ compared to the A*-based algorithm in TwinTwigJoin that solves the left-deep join plan, we arrive at optimality by settling the more challenging bushy join plan. We also show that it is beneficial to overlap edges among the join units. Given some practical relaxation, we are able to compute an optimal join plan that overlaps the join units with the same complexity as the non-overlapped case.

Despite the optimality guarantee in the proposed algorithms, the huge cost of subgraph enumeration can potentially affect the scalability to large graph. We therefore study two data compression techniques to further reduce the cost. The

first technique, called *compressed graph*, leverages the symmetric structure in the data graph. A set of nodes that have the *same neighborhoods* are aggregated into one *compressed node*, which transforms the original graph into a compressed graph. We then process the query on the compressed graph in order to save the cost of computing, transferring and maintaining the results associated with the compressed nodes. Ren et al. applied the technique in [RW15] to boost the labeled subgraph matching. However, their centralised algorithm cannot scale to web-scale real graph. In this work, we propose non-trivial distributed algorithms (on MapReduce) to construct the compressed graph and process queries on it. We first identify three kinds of compressed nodes, and use them to bind the compressed edges and construct the compressed graph. Then we extend the TwinTwigJoin algorithm to handle the compressed graph by leveraging the properties of compressed node in the computation of the matches of TwinTwig. Note that we can also adapt SEED to the compressed graph, but we focus on TwinTwigJoin in this article to deliver the intuitive idea. The second data compression technique, namely *clique compression*, utilizes the fact that any k-combination of the nodes in a large clique is a match of the k-clique. We first precompute and index all cliques in the data graph with sizes larger than a given threshold. When we are computing the matches of clique (as join unit) in SEED, we attempt to maintain the results that involves in the precomputed cliques in a compressed form as much as possible, which significantly reducing the cost of the algorithm. Clique compression can only applied to SEED where clique can be the join unit.

1.3 Contributions

We make the following contributions in this work.

(1) A general approach to process subgraph enumeration in the distributed context. We introduce a general decomposition-and-join approach for subgraph enumeration that can be implemented in a variety of general-purposed big data process engines. We formulate the distributed graph storage mechanism to determine what structures can serve the join units in the pattern decomposition.

(2) TwinTwigJoin*: Optimality in star-based left-deep join.* We first introduce the star-based join framework, which features a basic graph storage mechanism that only supports star as join unit and a left-deep join structure. A comprehensive cost model is introduced based on the ER model to evaluate the proposed algorithm, according to which we are able to prove that using TwinTwig as the join unit is enough to guarantee instance optimality in the star-based join framework, which motivates the TwinTwigJoin algorithm that uses TwinTwig instead of a general star as the join unit. We devise an A*-based algorithm to compute the optimal TwinTwig join plan, and explore three optimization strategies, namely, order-aware cost reduction, workload skew reduction, and early filtering, to further improve the TwinTwigJoin algorithm.

(3) SEED*: Optimality in graph-based bushy join* We explore the optimal graph-based bushy join to resolve the constraints of TwinTwigJoin, which results in the SEED algorithm. We show that it is storage-inefficient to use any general structure as the join unit, and then introduce the star-clique-preserved (SCP) storage mechanism that supports star and clique as the join units. In order to produce more realistic cost estimation, we base the cost analysis on the PR model rather than the ER model. Ultimately, we develop a dynamic-programming algorithm to compute an **optimal** bushy join plan. We also show that it is beneficial to overlap edges among the join units. Given some practical relaxation, we compute an optimal join plan that overlaps the join units with the same complexities as the non-overlapped

case.

(4) The data compression techniques. We propose two data compression techniques to further reduce the cost while processing subgraph enumeration. The first technique targets compressing the data graph by aggregating the nodes that have the same neighborhood into one single compressed node. The second technique aims at reducing the partial result set by using the set of nodes in a large cliques to represent all involved matches of a smaller clique.

1.4 Related Work

Dataflow Engines. The shared-nothing architecture has become a standard for large data processing nowadays, and there emerge a variety of shared-nothing data engines in both academy and industry. MapReduce was proposed by J.Dean et al. [DG04] to provide scalable and convenient big-data processing capabilities. Spark [MMJ+] improves MapReduce in multi-iteration tasks by offering in-memory computation. Dryad [IBY+] was proposed by M.Isard et al. as a general-purpose distributed execution engine for coarse-grain data-parallel applications. D.Halperin et al. proposed Myria [HTC+] targeting a distributed, shared-nothing big-data management system. These engines can all be used to implement TwinTwigJoin and SEED.

Subgraph Matching. Most subgraph matching approaches work in labeled context, where nodes (and/or edges) are assigned labels in both data and query graphs. For example, node labels in the neighborhood are used to filter unexpected candidates in [HS08] and [ZH10]. In [HLL13], the authors observed that a good matching order can significantly improve the performance of subgraph query. Lee et al. [LHKL12] provided an in-depth comparison of subgraph isomorphism

algorithms. Subgraph enumeration in a centralized environment has also been studied in exact and approximate settings. The exact solutions including [CN85] and [GK07] are not scalable to large data graphs. The approximate solutions [ADH+08, GRS10, ZKKM10] only estimate the count rather than locate all the subgraph instances.

Subgraph Matching in Cloud. Many recent works focused on solving subgraph matching in the cloud. Zhao et al. [ZKKM10] introduced a parallel color coding method for subgraph counting. Ma et al. [MCHW12] studied inexact graph pattern matching based on graph simulation in a distributed environment. Sun et al. [SWW+12] proposed a subgraph matching algorithm that uses node filtering to handle labeled graphs in the Trinity memory cloud. Recently, Shao et al. [SCC+14] developed PSgL to list subgraph instances in Pregel, which can be seen as a StarJoin-like algorithm and shall be proven to be worse than our TwinTwigJoin algorithm [LQLC15].

Subgraph Enumeration in MapReduce. Subgraph enumeration in MapReduce has attracted a lot of interests. Tsourakakis et al. [TKMF09] proposed an approximate triangle counting algorithm using MapReduce. Suri et al. [SV11] introduced a MapReduce algorithm to compute exact triangle counting. Afrati et al. [AFU13] proposed multiway join in MapReduce to handle subgraph enumeration. Plantenga [Pla13] introduced an edge join method in MapReduce which can be used for subgraph enumeration. In [FFF14], small cliques are enumerated using MapReduce, however the method can only be used to enumerate small cliques rather than any general pattern graphs.

1.5 Outline

Chapter 2 presents the preliminaries and formulates the problem. Chapter 3 introduces the general framework for the proposed algorithms. We introduce the star-based join framework, and show the instance optimality of the TwinTwigJoin algorithm in star-based join. In Chapter 5, we demonstrate how SEED resolves the constraints of TwinTwigJoin via the SCP graph storage mechanism and the optimal bushy join plan. Chapter 6 explores the two data compression techniques, and Chapter 7 concludes the whole article.

Chapter 2

Preliminaries

Given a graph g, we use $V(g)$ and $E(g)$ to denote the set of nodes and edges of g. For a node $\mu \in V(g)$, denote $\mathcal{N}(\mu)$ as the set of neighbors, and $d(\mu) = |\mathcal{N}(\mu)|$ as the degree of μ. A *subgraph* g' of g, denoted $g' \subseteq g$, is a graph that satisfies $V(g') \subseteq V(g)$ and $E(g') \subseteq E(g)$.

A *data graph* G is an undirected and unlabeled graph. Let $|V(G)| = N$, $|E(G)| = M$ (assume $M > N$), and $V(G) = \{u_1, u_2, \ldots, u_N\}$ be the set of data nodes. We define the following total order among the data nodes as:

Definition 2.1. *(**Node Order**) 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 $u \in V(G)$.

A *pattern graph* P is an undirected, unlabeled and connected graph. We let $|V(P)| = n$, $|E(P)| = m$, and $V(P) = \{v_1, v_2, \ldots, v_n\}$ be the set of pattern nodes.

We use $P = P' \cup P''$ to denote the merge of two pattern graphs, where $V(P) = V(P') \cup V(P'')$ and $E(P) = E(P') \cup E(P'')$.

Definition 2.2. ***(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 $1 \leq i \leq n$.

We say two graph g_i and g_j are isomorphic if and only if there exists a match of g_i in g_j, and $|V(g_i)| = |V(g_j)|$, $|E(g_i)| = |E(g_j)|$. The task of *Subgraph enumeration* is to enumerate all $g \in G$ such that g is isomorphic to P.

Remark 2.1. *An automorphism of P is an isomorphism from P to itself. Suppose there are A automorphisms of the pattern graph. If the number of enumerated subgraphs is s, then the number of matches of P in G is $A \times s$. Therefore, if P has only one automorphism, the problem of subgraph enumeration is equivalent to enumerating all matches (Definiton 2.2). Otherwise, there will be duplicate enumeration. In this work, for the ease of analysis, we will assume that the pattern graph P has only one automorphism, and focus on enumerating all matches of P in G.*

Details of resolving the duplication caused by automorphism will be discussed in Chapter 4.3.5.

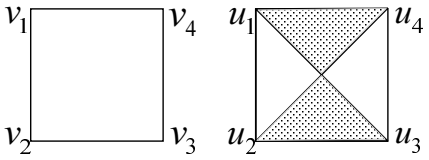


Figure 2.1: Pattern Graph P (Left) and Data Graph G (Right).

We use $R_G(P)$ to denote the matches of P in G, or simply $R(P)$ when the context is clear. Since a match is a one-to-one mapping from the pattern nodes to the data nodes, we regard $R(P)$ as a relation table with $V(P)$ as its attributes.

Example 2.1. *Figure 2.1 shows a square pattern graph P, 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).*

Problem Statement. Given a data graph G stored in the distributed file system, and a pattern graph P, the purpose of this work is to enumerate all matches of P in G (based on Definiton 2.2) in the distributed environment.

Remark 2.2. *For simplicity, we discuss the algorithm in MapReduce. However, all techniques proposed in this work are platform-independent, so it is seamless to implement the algorithm in any general-purpose distributed dataflow engine, such as Spark [MMJ+], Dryad [IBY+] and Myria [HTC+].*

Graph Models. In this work, we will depict the data graph using two graph models, based on which we can estimate the number of matches of any given pattern graph, so as to facilitate the computation of cost model and other theoretical analysis.

Erdös-Rényi Random (ER) Graph Model. We model the data graph as a Erdös-Reńyi Random (ER) graph according to [ER60], which is denoted as $\Re$. 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.

Power-Law Random (PR) Graph Model. We model the data graph as a power-law random (PR) graph according to [CLV03a], which is denoted as $\mathcal{G}$. Corresponding to the set of data nodes, we consider a non-decreasing degree sequence $\{w_1, w_2, \ldots, w_N\}$ that satisfies 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 [1]. For any pair of nodes u_i and u_j in a PR graph, the edge between u_i and u_j is independently assigned with probability

$$\mathsf{Pr}_{i,j} = w_i w_j \rho,$$

where $\rho = 1/\Sigma_{i=1}^{N} w_i$. It is easy to verify that the $\mathbb{E}[d(u_i)] = w_i$ for any $1 \leq i \leq N$ ($\mathbb{E}[\cdot]$ computes the expected value). We define the average degree as $w = (\Sigma_{i=1}^{N} w_i)/N$, and the expected maximum degree as w_{max}. In case that $\mathsf{Pr}_{i,j} \leq 1$ holds, we require $w_{max} \leq \sqrt{wN}$ [VL05]. As shown in [LQLC15], in real-life graphs, although there are nodes with degree larger than $\sqrt{wN}$, the intermediate results from these nodes are not the dominant parts in subgraph enumeration. In this work, if not otherwise specified, we simply let $w_{max} = \sqrt{wN}$. Given β, w, N and w_{max}, a degree sequence can be generated using the method in [VL05].

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.

[1] If not specially mentioned, β is set to $2 < \beta < 3$ in this work, a typical setting of β for real-life graphs [CLV03b, CSN09].

Summary of Notations. Table 2.1 summarizes the notations frequently used in this article.

Notations	Description
$V(g), E(g)$	The set of nodes and edges of a graph g
$\mathcal{N}(\mu), d(\mu)$	The set of neighbor nodes and the degree of $\mu \in V(g)$
G	The data graph
N, M	The number of nodes and edges in the data graph
u, u_i	An arbitrary data node and the data node with id i
P	The pattern graph
n, m	The number of nodes and edges in the pattern graph
v, v_i	An arbitrary pattern node and the patter node with id i
p_i	The join unit
$D(P) = \{p_0, p_1, \ldots, p_t\}$	The pattern decomposition
P_i	The i[-th] partial pattern, $P_i \subseteq P$
P_i^l, P_i^r	The left and right join patterns while processing P_i
f	A match of P in G
$R_G(P), R(P)$	The relation of the matches of P in G
$\Phi(G)$	The storage mechanism of G
G_u	The local graph of $u \in V(G)$, where $G_u \in \Phi(G)$
$\Re$	An Erdös-Rényi random graph
$\mathcal{G}$	A power-law random graph
β	The power-law exponent of $\mathcal{G}$
w_i	The expected degree of u_i in $\mathcal{G}$

Table 2.1: Notations frequently used in this article.

Chapter 3

A General Approach

In this section, we propose a general approach for subgraph enumeration, based on which we can describe all proposed algorithms in this work. The approach follows a decomposition-and-join manner. We will first introduce the concept of graph storage mechanism, which determines the join unit and hence the pattern decomposition. Consequently, we will discuss two join structures, left-deep join and bushy join, for join processing.

3.1 Execution Plan

Graph Storage. We solve the subgraph enumeration in a decomposition-and-join manner. Specifically, we first decompose the pattern graph into a set of sub-structures, called *join unit*, then we join the matches of these join units to gain the results.

To determine what structure can be the join unit, we first introduce the *graph storage mechanism*, which is defined as $\Phi(G) = \{G_u \,|\, u \in V(G)\}$, where $G_u \subseteq G$ is a connected subgraph of G with $u \in V(G_u)$, and it must satisfy that $(u, u') \in E(G_u)$ for all $u' \in \mathcal{N}(u)$. Each G_u is called the *local graph* of u. Specifically, the data

graph G is maintained in the distributed file system in the form of key-value pairs $(u; G_u)$ for each $G_u \in \Phi(G)$. We define the *join unit* as:

Definition 3.1. *(**Join Unit**) Given a data graph G and the graph storage $\Phi(G) = \{G_u \mid u \in V(G)\}$, a connected structure p is a join unit w.r.t. $\Phi(G)$, if and only if*

$$R_G(p) = \bigcup_{G_u \in \Phi(G)} R_{G_u}(p).$$

In other words, a join unit is a structure whose matches can be enumerated independently in each local graph $G_u \in \Phi(G)$. We further define *pattern decomposition* as:

Definition 3.2. *(**Pattern Decomposition**) Given a graph storage $\Phi(G)$, a pattern decomposition is denoted as $D = \{p_0, p_1, \ldots, p_t\}$, where $p_i \in P$ $(0 \leq i \leq t)$ is a join unit w.r.t. $\Phi(G)$ and $P = p_0 \cup p_1 \cup \cdots \cup p_t$.*

Join Plan. Given the decomposition $D = \{p_0, p_1, \ldots, p_t\}$ of P, we solve the subgraph enumeration using the following join:

$$R(P) = R(p_0) \bowtie R(p_1) \bowtie \cdots \bowtie R(p_t). \tag{3.1}$$

A *join plan* determines an order to solve the above join, and it processes t rounds of two-way joins. We denote P_i as the i[-th] *partial pattern* whose results are produced in the i[-th] round of the join plan. Obviously, we have $P_t = P$. The join plan is usually presented in a tree structure, where the leaf nodes are (the matches of) the join units, the internal nodes are the partial patterns.

A join tree uniquely specifies a join plan, and we use join tree and join plan interchangeably. If all internal nodes of the join tree have *at least* one join unit as its child, the tree is called a left-deep tree [1]. Otherwise it is called a bushy tree

[1]More accurately, it is the deep tree, which is further classified into the left-deep and right-deep tree. As it is insignificant to distinguish them here, we simply refer to the deep tree as left-deep.

[IK91]. Note that a left-deep tree is also a bushy tree.

Example 3.1. *Given a pattern decomposition $D(P) = \{p_0, p_1, p_2, p_3\}$, we present a left-deep tree and a bushy tree in Figure 3.1. Here we use triangle as the join unit. In the left-deep tree, we process $R(P_1^{ld}) = R(p_0) \bowtie R(p_1)$ in the first round, followed by $R(P_2^{ld}) = R(P_1^{ld}) \bowtie R(p_2)$, and finally $R(P) = R(P_2^{ld}) \bowtie R(p_3)$. Note that there involves a join unit in each round. In the bushy join, we first compute $R(P_1^b) = R(p_0) \bowtie R(p_1)$ and then $R(P_2^b) = R(p_2) \bowtie R(p_3)$, and finally $R(P) = R(P_1^b) \bowtie R(P_2^b)$ is processed. It is obvious that there is no join unit in the third round.*

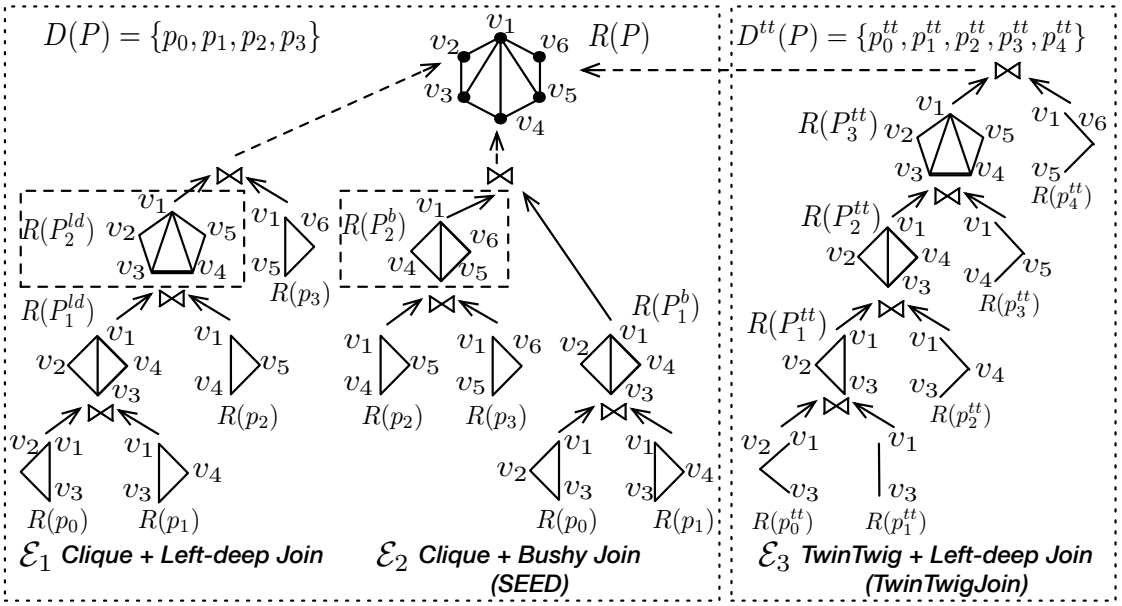


Figure 3.1: Different Join Trees.

It is worth noting that in the left-deep join, the pattern decomposition $D(P) = \{p_0, p_1, \ldots, p_t\}$ (with specified order) uniquely determines a join plan, in which the $i[-th]$ join is $R(P_i) = R(P_{i-1}) \bowtie R(p_i)$, where $P_0 = p_0$ and $P_i (i > 0) = p_0 \cup p_1 \cdots \cup p_i$.

Execution Plan. An *execution plan* of subgraph enumeration task, denoted as

$\mathcal{E} = (D, J)$, contains two parts - a *pattern decomposition* D and a *join plan* J. Consider an execution space Σ and a cost function $\mathcal{C}$ defined over Σ. We formulate the problem of *optimal execution plan* for solving subgraph enumeration as follows:

Definition 3.3. *(**Optimal Execution Plan**) An optimal execution plan for solving subgraph enumeration is an execution plan $\mathcal{E}_o = (D_o, J_o) \in \Sigma$ to enumerate P in G using Equation 3.1, such that,*

$$\mathcal{C}(\mathcal{E}_o) \text{ is minimized.}$$

3.2 Challenges

To pursuit the optimality for subgraph enumeration, we have to address multiple key challenges. Specifically,

- It is non-trivial to develop an effective graph storage mechanism. In star-based join (Chapter 4), we will use a basic graph storage that only supports star as the join unit. We resolve this constraint in the graph-based join (Chapter 5) by bring in extra edges to the simple local graph used in star join. However, the size of each local graph should not be too large for scalability consideration.

- A well-defined cost function is required to estimate the cost of each execution plan. In the subgraph enumeration problem, the tuples that participate in the joins are the matches of certain pattern graph, whose size is difficult to estimate, especially in a power-law graph.

- It is in general computationally intractable to compute an optimal join plan [JK84]. We will apply an easier-solving left-deep join in Chapter 4, which may render sub-optimal solution. In Chapter 5, we will further explore the

optimal bushy join plan - a much harder task given the larger searching space [IK91].

Chapter 4

TwinTwigJoin: Optimal Star-based Left-deep Join

In this chapter, we introduce the star-based join framework, in which we apply the basic graph storage mechanism and the left-deep join structure. We discuss three existing solutions, namely EdgeJoin, StarJoin and MultiwayJoin, among which the former two methods also follow the star join. According to a well-defined cost model and the ER model, we show that it is sufficient to guarantee instance optimality in the star-based join framework by using TwinTwig, instead of a general star as the join unit, which inspires our TwinTwigJoin algorithm. We explore three optimization strategies to improve the performance of the algorithm. Ultimately the performance studies are conducted to demonstrate the effectiveness of the proposed techniques.

4.1 Star-based Join Framework

The star-based join framework implements the basic graph storage mechanism and the left-deep join structure in the general approach (Chapter 3), and can generalize

the EdgeJoin, StarJoin and TwinTwigJoin algorithms.

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

function SubgraphEnum-Star (G, P)
compute a graph decomposition $\{p_0, p_1, \ldots, p_t\}$ of P, where each join unit is a star;
for $i = 1$ **to** t **do**
    $R(P_i) \leftarrow R(P_{i-1}) \bowtie R(p_i)$; (using map^i and reduce^i)
return $R(P_t)$;
function map^i(**key**: $\emptyset$; **value**: Either a match $f \in R(P_{i-1})$ when $i > 1$ or a local graph $G_u^0 \in \Phi^0(G)$)
$\{v_{k_1}, v_{k_2}, \ldots, v_{k_l}\} \leftarrow V(P_{i-1}) \cap V(p_i)$;
if $i = 1$ **then**
    $R_(G_u)(P_0) \leftarrow$ all matches of P_0 in G_u;
    forall the match $f \in R(G_u)(P_0)$ **do**
        output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_l})); f)$;
if Value is a match $f \in R(P_{i-1})$ **then**
    output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_l})); f)$;
else
    $R(G_u)(p_i) \leftarrow$ all matches of p_i in G_u;
    forall the match $h \in R_u(p_i)$ **do**
        output $((h(v_{k_1}), h(v_{k_2}), \ldots, h(v_{k_l})); h)$;
function reduce^i(**key**:$r = (u_{k_1}, u_{k_2}, \ldots, u_{k_s})$; **value**: $F = \{f_1, f_2, \ldots\}$, $H = \{h_1, h_2, \ldots\}$)
forall the $(f, h) \in (F \times H)$ s.t. $(f - r) \cap (h - r) = \emptyset$ **do**
    output $(\emptyset; f \cup h)$;

Basic Graph Storage. We denote the basic graph storage mechanism as

$$\Phi^0(G) = \{G_u^0 \mid u \in V(G)\},$$

where $V(G_u^0) = \{u\} \cup \mathcal{N}(u)$ and $E(G_u^0) = \{(u, u') | u' \in \mathcal{N}(u)\}$. Star is the join unit regarding $\Phi^0(G)$, as the matches of a k-star (a star of k edges) rooted at a certain node u can be computed by enumerating the k-combinations from $\mathcal{N}(u)$. It is also obvious that star is the only join unit for $\Phi^0(G)$ that satisfies Definiton 3.1.

Left-deep join structure. The star-based join framework applies the left-deep join structure. As we mentioned in Chapter 3, the pattern decomposition (with fixed order) itself can determine a left-deep join plan. Consequently, in the rest of the chapter, we will use pattern decomposition and the join plan interchangeably. Also, we redefine the pattern decomposition and partial pattern in this chapter for the ease of presentation.

Definition 4.1. ***(Pattern Decomposition)*** *Given a pattern graph P, a pattern decomposition of P, $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$ is a **disjoint** partition of the edges of P, such that p_i $(0 \leq i \leq t)$ 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 4.2. ***(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 join unit p_i shares at least a common node with the partial pattern P_{i-1} for any $1 \leq i \leq t$. Note that the pattern decomposition used here is a disjoint partition of the pattern edges, which

is not a constraint in Definiton 3.2. As a matter of fact, it is beneficial to allow edge overlaps in the decomposition when more complex structures are considered as the join unit (details in Chapter 5.2.3).

Algorithm Overview. We show the star-based join framework in Algorithm 1. We first compute a pattern decomposition $\{p_0, p_1, \ldots, p_t\}$ of P, where each join unit is a star (details in Chapter 4.3.4). The decomposition itself indicates an optimal left-deep join plan (line 2). Then the algorithm is processed in t MapReduce rounds. Each round computes the result set $R(P_i)$ by joining $R(P_{i-1})$ with $R(p_i)$ (line 4) using MapReduce via map^i and reduce^i.

(Function map^i*)*:. According to the left-deep join, the input of map^i is either a match $f \in R(P_{i-1})$ if $i > 1$, or a local graph $G_u^0 \in \Phi^0(G)$ (line 6). We first calculate the join key (line 7). If $i = 1$, we need to compute the matches of $P_0 = p_0$, $R_{G_u}(P_0)$, based on node u and its neighbours $\mathcal{N}(u)$. We output each such match (as a match in $R(P_0)$) along with the corresponding join key (lines 8-10). As p_0 is a star, we can compute its matches by enumerating the node combinations in $\mathcal{N}(u)$ (G_u). 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 12). Otherwise, we compute the matches of p_i in G_u, as we do when we compute p_0 (lines 14-16).

(Function reduce^i*)*: The set of key-value pairs with the same key $r = (u_{k_1}, u_{k_2}, \ldots, u_{k_l})$ are processed using the same function reduce^i. There are two types of values, $F = \{f_1, f_2, \ldots\}$ and $H = \{h_1, h_2, \ldots\}$, generated by $R(P_{i-1})$ and $R(p_i)$ respectively. For each $(f, h) \in (F \times H)$ that shares the same join key, we output $f \cup h$ with the condition that $(f - r) \cap (h - r) = \emptyset$ to avoid node conflict (refer to the conflict freedom condition in Definiton 2.2)(lines 18-19).

Example 4.1. *In Figure 4.1, we decompose the pattern graph into* $\{p_0, p_1, p_2\}$.

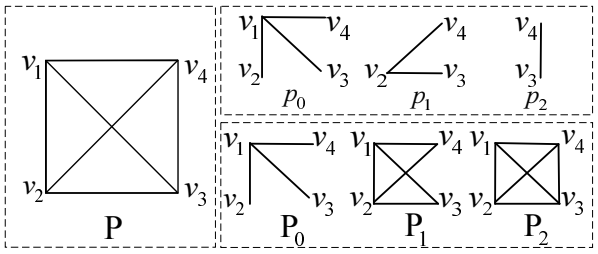


Figure 4.1: The pattern decomposition and the corresponding partial patterns.

The corresponding partial patterns P_0, P_1, and P_2 are also presented. Based on the left-deep join plan, 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.

Assumptions. To facilitate our theoretical analysis, we consider the following assumptions in this chapter.

- A_1: The data graph is a random graph.
- A_2: The algorithm follows the star-based join framework.
- A_3: The data graph is sparse; more specifically, the average degree $d = 2M/N < \sqrt{N}$.

4.2 Existing Work

In this subchapter, we introduce three state-of-the-art algorithms for subgraph enumeration: EdgeJoin, StarJoin, and MultiwayJoin. Both EdgeJoin and StarJoin follow the star-based join framework with different pattern decomposition strategies. MultiwayJoin uses a new strategy that enumerates all subgraphs using only one

MapReduce round by duplicating the data edges. In the following, we introduce EdgeJoin and StarJoin in Chapter 4.2.1 and MultiwayJoin in Chapter 4.2.2.

4.2.1 Star-based Join

Algorithm EdgeJoin. The EdgeJoin Algorithm is proposed by Plantenga [Pla13]. In EdgeJoin, each pattern graph P is decomposed into $\{p_0, p_1, \cdots, p_t\}$ where each p_i is an edge in $E(P)$ (note that an edge is a special star). Then the data edges are joined to produce the results via a specific order. The EdgeJoin Algorithm has two drawbacks. Firstly, it may generate a large number of intermediate matches. Secondly, it needs $t = m-1$ MapReduce rounds, which are too many for a complex pattern graph. We explain the two drawbacks using the following example.

Example 4.2. *For the square given in Example 2.1, 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 three execution rounds, and the partial pattern* P_3 *is a path of length* 3*, whose result set can be enormously large. A better strategy is to decompose* P *into two parts:* $p_0 = \{(v_1, v_2), (v_2, v_3)\}$ *and* $p_1 = \{(v_3, v_4), (v_4, v_1)\}$*, which can be processed in only one MapReduce round, and the size of the intermediate matches is not large, relative to the size of the final result* $R(P)$.

Algorithm StarJoin. The StarJoin algorithm decomposes the pattern graph into a set of stars using the the strategy proposed by Sun et al. [SWW+12]. 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 4.3. ***(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* $1 \leq i \leq t$.

From a selected initial node v_{k_0}, the star decomposition iteratively extracts star composed with a node and its neighbors in the residual pattern graph until the pattern graph becomes empty. Compared to EdgeJoin, StarJoin can often largely reduce the number of execution rounds, however, StarJoin still suffers from the scalability issue due to the large number of intermediate results generated when evaluating a star with many edges, as we mentioned in Chapter 1.

Essentially, the PSgL algorithm [SCC+14], following a BFS strategy, can be considered as a StarJoin algorithm [LQLC15] that processes the joins between the matches of the star rooted on current visiting node and the partial subgraph instances obtained from the previous step. As a result, PSgL also suffers the above issue of StarJoin.

4.2.2 Multiway Join

The MultiwayJoin algorithm was proposed by Afrati et al. [AFU13], and it enumerates subgraphs in only one MapReduce round. In the map phase, each edge in G is duplicated several times, and each duplication is sent to a certain reducer. In the reduce phase, each reducer computes its matches independently according to the set of edges received.

The map and reduce functions of MultiwayJoin is shown in Algorithm 2. Let h_i $(1 \leq i \leq n)$ be a hash function that hashes an arbitrary node $u \in V(G)$ to an integer in the range of $[1, b_i]$ (line 3). In the map phase, for each edge $(u, u') \in E(G)$, it traverses each edge $(v_i, v_j) \in E(P)$ (lines 4-5) to be matched by (u, u') in a certain reducer. For each such candidate edge to be matched, it enumerates all possible combinations $(x_1, x_2, \cdots, x_n)$ with $x_i = h_i(u)$, $x_j = h_j(u')$, and $x_k \in \{1, 2, \cdots, b_k\}$ for all $1 \leq k \leq n$ with $k \neq i$ and $k \neq j$ (line 6). Each of the combination is

Algorithm 2: MultiwayJoin(data graph G, pattern graph P)

Input : G, The data graph,

P, The pattern graph.

Output : $R(P)$, all Matches of P in G.

1 **function** map(**key**: $\emptyset$; **value**: G)

2 $X_i \leftarrow \{1, 2, \cdots, b_i\}$ for any $1 \leq i \leq n$;

3 $h_i \leftarrow$ a hash function that maps $v \in V(G)$ to $x \in X_i$ $(1 \leq i \leq n)$;

4 **forall the** $(u, u') \in E(G)$ **do**

5 **forall the** $(v_i, v_j) \in E(P)$ **do**

6 **forall the** $(x_1, x_2, \cdots, x_n)$ s.t. $x_i = h_i(u)$, $x_j = h_j(u')$, and $x_k \in X_k$ (for all $1 \leq k \leq n$, $k \neq i$, $k \neq j$) **do**

7 output a key-value pair with key $(x_1, x_2, \cdots, x_n)$ and value $(u, u') \rightarrow (v_i, v_j)$;

8 **function** reduce(**key**: $(x_1, x_2, \cdots, x_n)$; **value**: $G' = \{(u_{k_1}, u'_{k_1}) \rightarrow (v_{i_1}, v_{j_1}), (u_{k_2}, u'_{k_2}) \rightarrow (v_{i_2}, v_{j_2}), \cdots\}$)

9 compute all matches for P based on candidate edges in G';

sent to the reducer represented by key $(x_1, x_2, \cdots, x_n)$ with value $(u, u') \rightarrow (v_i, v_j)$ indicating that (u, u') is a candidate edge to be matched by (v_i, v_j) in the reducer (line 7). In the reduce phase, for each key $(x_1, x_2, \cdots, x_n)$ it simply collects all the candidate edges to match each $(v_i, v_j) \in E(P)$, and computes all the matches for P according to such candidate edges (lines 8-9). How to select each b_i for $1 \leq i \leq n$ to minimize the total communication cost is discussed in [AFU13]. It is easy to prove that each match $(u_{k_1}, u_{k_2}, \cdots, u_{k_n})$ will only be generated in the reducer with key $(h_1(u_{k_1}), h_2(u_{k_2}), \cdots, h_n(u_{k_n}))$.

Example 4.3. *Let P be a triangle (v_1, v_2, v_3). In this case, the optimal b_i assign-*

ment is $b_1 = b_2 = b_3 = b$ according to [AFU13]. In Algorithm 2, for each edge $(u, u') \in E(G)$, it is sent to the reducers in the following three groups:

- *(u, u') matches (v_1, v_2): (u, u') is sent to b reducers with key $(h(u), h(u'), x_3)$ for $1 \leq x_3 \leq b$.*
- *(u, u') matches (v_2, v_3): (u, u') is sent to b reducers with key $(x_1, h(u), h(u'))$ for $1 \leq x_1 \leq b$.*
- *(u, u') matches (v_1, v_3): (u, u') is sent to b reducers with key $(h(u), x_2, h(u'))$ for $1 \leq x_2 \leq b$.*

As a result, each edge is duplicated for $3b - 2$ times, and the total number of reducers is b^3. In this way, each triangle in G that matches P is guaranteed to be generated in one of the reducers.

Cost Analysis. It is shown in [AFU13] 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, in which the optimal b_i assignment is $b_1 = b_2 = \cdots = b_n = b$ according to [AFU13]. Using MultiwayJoin, the number of duplications for each edge $(u, u') \in E(G)$ is $\Theta(m \cdot b^{n-2}) = \Theta(n^2 \cdot b^{n-2})$. The number of reducers is $\Theta(b^n)$. Thus, the total number of edge duplications are $\Theta(M \cdot n^2 \cdot b^{n-2})$ and the edge duplications received be each reducer is $\Theta(\frac{M \cdot n^2 \cdot b^{n-2}}{b^n}) = \Theta(M \cdot \frac{n^2}{b^2})$. 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 handling either a large data graph or a complicated pattern graph. Similar result can be derived when P is a general graph.

4.3 TwinTwigJoin Algorithm

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 subchapter, we propose a new algorithm TwinTwigJoin that also follows the star join with a new pattern decomposition strategy, namely, TwinTwig decomposition. We first introduce the TwinTwig decomposition strategy, and analyze its optimality based on the ER graph model. Then we propose an optimal TwinTwig decomposition algorithm based on the A* framework. Finally, we discuss how to adapt the TwinTwigJoin to the power-law graph.

4.3.1 TwinTwig Decomposition

Definition 4.4. *(*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.*

TwinTwigJoin is a tradeoff between EdgeJoin and StarJoin. 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 4.5. ***(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, based on which we can prove the instance optimality of TwinTwigJoin in the star-based join framework under the aforementioned assumptions.

4.3.2 Cost Analysis

Cost Model. 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 the algorithm, including I/O cost, communication cost, computational cost, number of MapReduce rounds,

and workload balancing. We hence consider an overall cost $\mathcal{C}$ as follows:

$$\begin{aligned}\mathcal{C} &= \sum_{i=1}^{t}(|\mathcal{M}_i| + |\mathcal{S}_i| + |\mathcal{R}_i|) \\ &= 3\sum_{i=1}^{t}|R(P_i)| + |R(P_0)| + \sum_{i=1}^{t}|R(p_i)| + t|E(G)| - 2|R(P_t)| \qquad (4.1)\\ &= 3\sum_{i=1}^{t}|R(P_i)| + \sum_{i=0}^{t}|R(p_i)| + t|E(G)| - 2|R(P_t)|.\end{aligned}$$

Obviously, $\mathcal{C}$ is a comprehensive measurement of I/O cost, communication cost and computational cost, and it also implies the impact of the number of MapReduce rounds. Note that the last term $2|R(P_t)| = 2|R(P)|$ is independent of the decomposition strategy, thus it can be removed from the cost function. Therefore, given any pattern decomposition $\mathcal{D} = \{p_0, p_1, \ldots, p_t\}$, the cost function, denoted as $\mathsf{cost}(\mathcal{D})$, can be defined as

$$\mathsf{cost}(\mathcal{D}) = 3\sum_{i=1}^{t}|R(P_i)| + \sum_{i=0}^{t}|R(p_i)| + t|E(G)|. \qquad (4.2)$$

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

$$\mathsf{cost}(\mathcal{D}_i) = 3\sum_{j=1}^{i}|R(P_j)| + \sum_{j=0}^{i}|R(p_j)| + i|E(G)|. \qquad (4.3)$$

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

$$\mathsf{cost}(\mathcal{D}_i) = \mathsf{cost}(\mathcal{D}_{i-1}) + 3|R(P_i)| + |R(p_i)| + |E(G)|. \qquad (4.4)$$

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

Graph Model. We apply two graph models, namely the ER model [ER60], and the PR model [ACL00], for the purpose of evaluating the cost of different decomposition strategies. 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 graph in Chapter 4.4.

Recall that a match f of P in G should satisfy two conditions, namely, conflict freedom and structure preservation (refer to Definiton 2.2). With the two conditions, we can derive the following lemma on the expected number of matches of P in a random graph. In the following, for ease of analysis, when calculating the cost using ER model, we relax the conflict free condition of a match to allow duplicated nodes in a match. Thus, the number of matches calculated is an upper bound of the actual number of matches. Under such an assumption, we can derive the following lemmas on the number of matches of pattern graph P a random graph.

Lemma 4.1. *Given a random graph $\Re$ and a pattern graph P, if P is a tree, we have $(1-\epsilon) \times \frac{(2M)^{n-1}}{N^{n-2}} \leq |R(P)| \leq \frac{(2M)^{n-1}}{N^{n-2}}$, where $\epsilon = \frac{n^2}{N}$.*

Proof. We first prove $(1-\frac{n}{N})^n \times \frac{(2M)^{n-1}}{N^{n-2}} \leq |R(P)| \leq \frac{(2M)^{n-1}}{N^{n-2}}$ by induction. When P is an edge, i.e., $n = 2$, obviously, we have $|R(P)| = \frac{(2M)^{n-1}}{N^{n-2}} = 2M$. Suppose when $n = k$, it holds that

$$(1-\frac{k}{N})^k \times \frac{(2M)^{k-1}}{N^{k-2}} \leq |R(P)| \leq \frac{(2M)^{k-1}}{N^{k-2}}.$$

When $n = k+1$, P can be formed by combining a tree P' with k nodes and an edge (v, v') with $v \in V(P')$ and $v' \notin V(P')$. Thus, given any match f' of P', v' can match any node in $V(G) - f'$ with probability ω. It follows that

$$|R(P)| = |R(P')| \times (N - |V(P')|) \times \omega.$$

On the one hand,

$$\begin{aligned} |R(P)| = |R(P')| \times (N-k) \times \omega &\geq (1-\frac{k}{N})^{k+1} \times \frac{(2M)^k}{N^{k-1}} \\ &\geq (1-\frac{k+1}{N})^{k+1} \times \frac{(2M)^k}{N^{k-1}}; \end{aligned}$$

on the other hand,

$$|R(P)| = |R(P')| \times (N-k) \times \omega \leq |R(P')| \times N \times \omega \leq \frac{(2M)^{n-1}}{N^{n-2}}.$$

Therefore, by induction, we prove $(1-\frac{n}{N})^n \times \frac{(2M)^{n-1}}{N^{n-2}} \leq |R(P)| \leq \frac{(2M)^{n-1}}{N^{n-2}}$, since $(1-\frac{n}{N})^n \geq 1-\frac{n^2}{N} = 1-\epsilon$, Lemma 4.1 holds. □

Lemma 4.2. *Given a random graph $\Re$ and a pattern graph P, if P is a connected graph, we have $(1-\epsilon) \times \frac{(2M)^m}{N^{2m-n}} \leq |R(P)| \leq \frac{(2M)^m}{N^{2m-n}}$, where $\epsilon = \frac{n^2}{N}$.*

Proof. Since P is a connected graph, we can obtain a spanning tree P' of P. According to Lemma 4.1:

$$(1-\epsilon) \times \frac{(2M)^{n-1}}{N^{n-2}} \leq |R(P')| \leq \frac{(2M)^{n-1}}{N^{n-2}}.$$

Given a match $(u_{k_1}, u_{k_2}, \ldots, u_{k_n})$ of P', for any edge $(v_i, v_j) \in E(P) - E(P')$, with probability ω, $(u_{k_i}, u_{k_j}) \in E(G)$. There are totally $m-n+1$ edges in $E(P) - E(P')$, thus we have

$$|R(P)| = |R(P')| \times \omega^{m-n+1}.$$

Therefore, $(1-\epsilon) \times \frac{(2M)^m}{N^{2m-n}} \leq |R(P)| \leq \frac{(2M)^m}{N^{2m-n}}$. □

Remark 4.1. *In practice, when the graph is large, $\epsilon = \frac{n^2}{N}$ is close to 0. Therefore, we can use $\frac{(2M)^m}{N^{2m-n}}$ to estimate $|R(P)|$ with a small bounded error.*

4.3.3 Instance Optimality of TwinTwigJoin

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

Lemma 4.3. *Consider a random graph $\Re$, and 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*

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

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

$$|R(P_i)| = |R(P_{i-1})| \times (\frac{2M}{N^2})^{\Delta m_i} \times N^{\Delta n_i}. \tag{4.5}$$

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

- ($|E(p_i)| = 1$ 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 $|R(P_i)| \leq |R(P_{i-1})| \times \frac{(2M)^2}{N^3}$. As a result, $|R(P_i)| \leq |R(P_{i-1})| \times \frac{(2M)^2}{N^3} \leq |R(P_{i-2})| \times (\frac{(2M)^2}{N^3})^2 \leq \ldots \leq |R(p_0)| \times (\frac{(2M)^2}{N^3})^i$. □

Corollary 4.1. *Consider a random graph* $\Re$ *and 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*

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

Proof. 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 4.1 holds according to Lemma 4.3. □

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

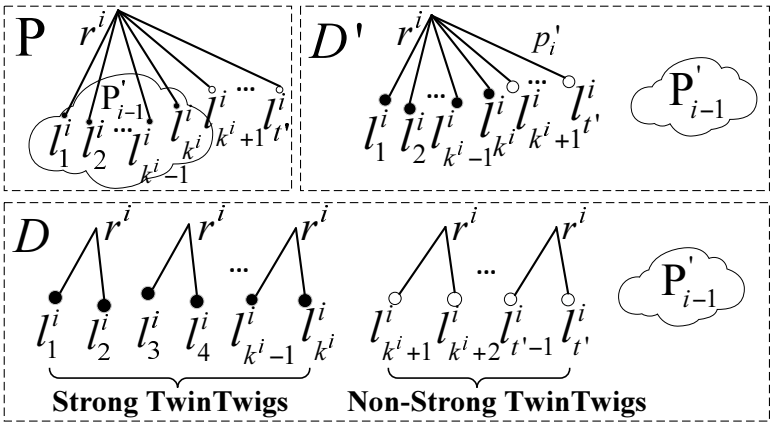


Figure 4.2: Constructing the TwinTwig decomposition $\mathcal{D}$ based on a certain star decomposition $\mathcal{D}'$.

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 Figure 4.2.

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

Theorem 4.1. *Consider a random graph $\Re$, a pattern graph P and 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* $\mathsf{cost}(\mathcal{D}) \leq \Theta(\mathsf{cost}(\mathcal{D}'))$.

Proof. For any pattern decomposition $\mathcal{D}$, we divide $\mathsf{cost}(\mathcal{D}) = 3\sum_{i=1}^{t} |R(P_i)| + \sum_{i=0}^{t} |R(p_i)| + t|E(G)|$ (Equation 4.2) into two parts:

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

Accordingly, we divide the proof into two parts:

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

- $\mathsf{cost}_1(\mathcal{D}^i) \leq \Theta(t'_i \cdot d^2 \cdot N)$. According to Lemma 4.2, we know that each pattern $p^i_j \in D^i$ is a TwinTwig with $|R(p^i_j)| \leq \frac{(2M)^2}{N} = \Theta(d^2 \cdot N)$. Hence, we have

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

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

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

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

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

$$\mathsf{cost}_2(\mathcal{D}') = \Theta(\sum_{i=1}^{t'}(|R(P'_{i-1})| + |R(P'_i)|)). \tag{4.6}$$

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

$$\begin{aligned} \mathsf{cost}^i_2(\mathcal{D}) = |R(P'_{i-1})| + |R(P'_{i-1} \cup p^i_1)| + \\ \cdots + |R(P'_{i-1} \cup p^i_1 \cup \cdots \cup p^i_{t_i-1})| + |R(P'_i)|. \end{aligned} \tag{4.7}$$

Recall that there exists a k_i such that, when $1 \leq j \leq k_i$, p^i_j is a strong TwinTwig, and when $k_i < j \leq t_i$, 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 4.3 and Corollary 4.1, when j increases, the size of the j-th term decreases exponentially with a rate $\leq \frac{(2M)^2}{N^3} < 1$, 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 Equation 4.5, 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 4.1 holds. □

4.3.4 Optimal Execution Plan

Based on the Theorem 4.1, we know that TwinTwigJoin (with TwinTwig decomposition) assures instance optimality in the star-based join framework. We discuss an A*-based algorithm to compute an optimal TwinTwig decomposition, which corresponds to an optimal execution plan as defined in Definiton 3.3

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 4.2), we need to define a cost function $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$, which is the cost lower bound for any TwinTwig decomposition of P expanded from $\mathcal{D}_i$. We compute $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ using dynamic programming. Given a partial pattern P_i, we use $\underline{\Delta\mathsf{cost}}(P_i, \Delta m, \Delta n)$ to denote the lower bound of the increased cost when adding any Δm edges and Δn nodes into the partial pattern P_i. Let $\mathsf{card}(m, n) = |R(P)|$ be the number of matches of any connected pattern graph P with m edges and n nodes, according to Lemma 4.2, we have

$$\mathsf{card}(m, n) = (2M)^m / N^{2m-n}. \tag{4.8}$$

In the dynamic-programming algorithm, the initial state is $\underline{\Delta\mathsf{cost}}(P_i, 0, 0) = 0$,

and according to Equation 4.4, 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

$$\begin{aligned}\underline{\mathsf{cost}}(\mathcal{D}_i, P) = \mathsf{cost}(\mathcal{D}_i) + \\ \underline{\Delta\mathsf{cost}}(P_i, |E(P)| - |E(P_i)|, |V(P)| - |V(P_i|).\end{aligned} \tag{4.9}$$

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

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

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

The Algorithm. The A* algorithm to compute the optimal decomposition is shown in Algorithm 3. 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

Algorithm 3: OptExecPlan-TwinTwig(data graph G, pattern graph P)

Input : G, The data graph,

P, The pattern graph.

Output : The optimal TwinTwigJoin plan for 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 The optimal left-deep join plan determined by D';
```

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 TwinTwig decomposition (line 5) that determines the optimal left-deep join plan.

Lemma 4.4. *The space complexity and time complexity of Algorithm 3 are* $O(2^m)$ *and* $O(\overline{d} \cdot m \cdot 2^m)$ *respectively, where* $\overline{d} = \max_{v \in V(P)} d(v)$.

Proof. 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

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

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

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

□

Discussion. In practice, the processing time for Algorithm 3 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.

4.3.5 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 [GK07]. 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 4.6. ***(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 2.2.*
- *(Structure Preservation) The same as that in Definiton 2.2.*
- *(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 2.2, a new order-preservation constraint is added in the new definition of a match.

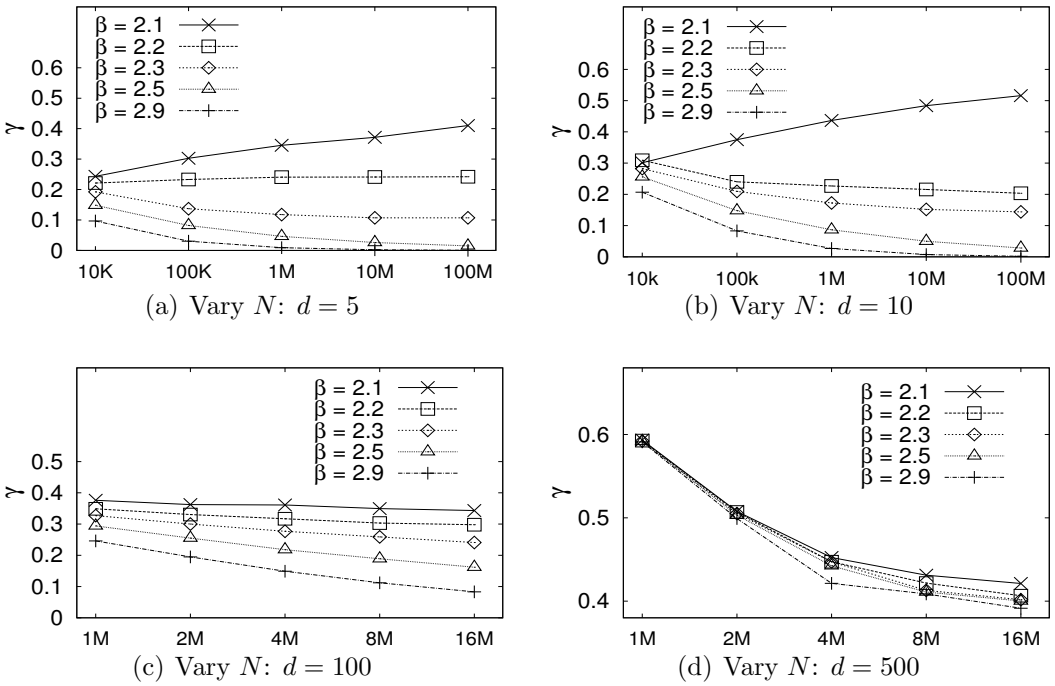
Example 4.4. *The square given in Example 2.1 has 8 automorphisms. Thus, each result subgraph will be duplicated 8 times using Algorithm 1. For example, the 8 matches (u_1, u_2, u_3, u_4), (u_2, u_3, u_4, u_1), (u_3, u_4, u_1, u_2), (u_4, u_1, u_2, u_3), (u_4, u_3, u_2, u_1), (u_3, u_2, u_1, u_4), (u_2, 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 9, line 15), we make sure that each match satisfies the *order preservation* constraint. In the reduce^i phase, in line 19, we only output those $f \cup h$ that satisfy the *order preservation* constraint. In Chapter 4.5.1, we will discuss how to use the partial order to further optimize the pattern decomposition.

Note that all proposed algorithms based on the general approach (including SEED) can use the above method to remove duplication caused by the isomorphism of the pattern graph, thus we do not repeat in detail afterwards.

4.4 Handling Power-Law Graphs

In this subchapter, we adapt TwinTwigJoin to the power-law graphs, by showing the instance optimality of TwinTwigJoin in the power-law random graph.



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

(c) Vary N: $d = 100$ (d) Vary N: $d = 500$

Figure 4.3: The values of γ in different parameter combinations.

Recall that in the PR graph, the edge between u_i and u_j is independently assigned with probability $P_{i,j} = w_i w_j \rho$, where $\rho = 1/\sum_{i=1}^{N} w_i = 1/2M$. We engage the small-degree assumption A_5 in this model as follows:

$$A_4 : d_{max} \leq \sqrt{N}.$$

Though this assumption may not be satisfied in some real graphs, in the experiment (Chapter 4.6), we show the intermediate results from the nodes with degree $\leq \sqrt{N}$ play a dominant role in the total intermediate results.

Instance Optimality. In order to show the instance optimality, we will prove that Theorem 4.1 holds in a power-law random graph under the small-degree assump-

tion A_4, following the same proof structure as that in the proof of Theorem 4.1. Similarly, 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 Equation 4.6 and Equation 4.7, and then prove the two cases, namely, S_1: the size of the results decreases after joining a strong TwinTwig; S_2: the size of the results increases after joining a non-strong TwinTwig. The detailed proof is as follows.

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

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

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:

$$|R(P_i)| = \gamma |R(P_{i-1})| \tag{4.10}$$

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:

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

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:

$$\gamma = d^2 \times \rho^2 \sum\nolimits_{i=1}^{N} w_i^2 = \frac{\sum_{i=1}^{N} w_i^2}{N^2} \tag{4.11}$$

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:

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

Therefore, γ can be calculated as:

$$\gamma = \rho^2 (\sum\nolimits_{i=1}^{N} w_i^2)^2 \times \rho^2 \sum\nolimits_{i=1}^{N} w_i^2 = \frac{(\sum_{i=1}^{N} w_i^2)^3}{(\sum_{i=1}^{N} w_i)^4} \tag{4.12}$$

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 [VL05], and thus γ can be calculated via Equation 4.12. The results are shown in Figure 4.3, 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:

$$\begin{aligned} \gamma &= \mathbb{E}[\sum\nolimits_{u',u'' \in V(G)} d(u)d(u')\rho \times d(u)d(u'')\rho] \\ &= \mathbb{E}[d(u)^2]\rho^2 \sum\nolimits_{i,j=1}^{N} w_i w_j = \mathbb{E}[d(u)^2] = \sum\nolimits_{i=1}^{N} w_i^2/N \end{aligned} \tag{4.13}$$

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

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

Optimal Decomposition. We show how to compute the optimal TwinTwig decomposition using A* for power-law random graph. Recall that Algorithm 3 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 Equation 4.4, 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 Equation 4.10, 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 Equation 4.11, Equation 4.12, and Equation 4.13, respectively. In this way, Algorithm 3 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 4.4.

4.5 Optimization Strategies

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

4.5.1 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 Equation 4.2, 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 $V(G)$ as in Definiton 2.1.

Given the new total order for $V(G)$, for any $u \in V(G)$, we let $\mathcal{N}^+(u) = \{u' \mid u' \in \mathcal{N}(u), u \prec u'\}$ and $\mathcal{N}^-(u) = \{u' \mid u' \in \mathcal{N}(u), u' \prec u\}$. We then define $d^+(u) = |\mathcal{N}^+(u)|$ and $d^-(u) = |\mathcal{N}^-(u)|$, and $d^+_{max} = \max_{u \in V(G)} d^+(u)$ and $d^-_{max} = \max_{u \in V(G)} d^-(u)$.

For a two-edge TwinTwig $p = \{(v, v_1), (v, v_2)\}$, we consider the following three types of orders:

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

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

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

where α is the *arboricity* of the graph G and $\alpha \leq d^+_{max} \leq d^-_{max}$ according to [CN85]. 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 4.5. *Figure 4.4 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 $\mathcal{D}^2$.*

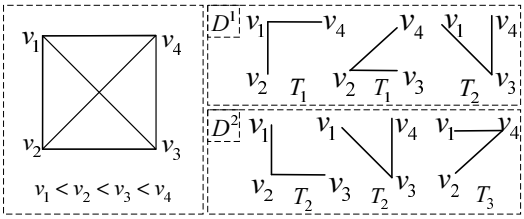


Figure 4.4: The order-aware decomposition of a 4-Clique.

Order-aware TwinTwig **Decomposition**. We discuss how to modify Algorithm 3 for TwinTwig decomposition by taking the partial order into consideration. Recall that Algorithm 3 only depends on the cost function $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ (Equation 4.9) for any partial TwinTwig decomposition $\mathcal{D}_i$, and $\underline{\mathsf{cost}}(\mathcal{D}_i, P)$ is calculated based on $\mathsf{cost}(\mathcal{D}_i)$ and $\underline{\Delta\mathsf{cost}}(P_i, \Delta m, \Delta n)$, both of which are originated from Equation 4.2. 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 Equation 4.5, $|R(P_i)|$ can be calculated as:

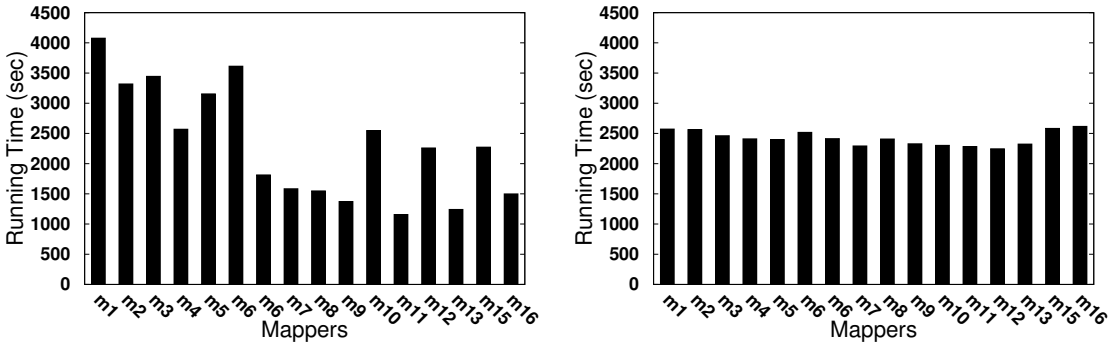
$$|R(P_i)| = |R(P_j)| \times (\frac{2M}{N^2})^{|E(P_i)|-|E(P_j)|} \times N^{|V(P_i)|-|V(P_j)|} \tag{4.14}$$

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 3) 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.

4.5.2 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.



(a) No Workload Balancing (b) After Workload Balancing

Figure 4.5: The Effect of Workload Balancing

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

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

Example 4.6. *Figure 4.5 shows the workload distribution of* 16 *mappers of* TwinTwigJoin *when evaluating the square pattern graph P shown in Figure 2.1 in the orkut dataset (see Chapter 4.6 for the detailed description of the dataset). Obviously, by considering workload balancing, the processing time is reduced by nearly half, and all mappers stop almost at the same time.*

4.5.3 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 [Blo70] 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 $\mathcal{B}(S)$, 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 $\mathcal{B}(E(G))$ in every machine of the system, and we use the bloom filter $\mathcal{B}(E(G))$ 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 $\mathcal{B}(E(G))$ 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 $\mathcal{B}(E(G))$ with probability $1 - fp$.

Obviously, early filtering does not affect the correctness of Algorithm 4 since only invalid partial patterns are pruned by the bloom filter $\mathcal{B}(E(G))$. Note that early filtering can be applied for all the three algorithms EdgeJoin, StarJoin, and TwinTwigJoin.

Example 4.7. *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 Equation 4.2, 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 $\mathcal{B}(E(G))$ 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.*

4.6 Performance Studies

In this subchapter, we show our experimental results for TwinTwigJoin algorithm. 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 4.1) 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).

Table 4.1: Datasets used in the TwinTwigJoin experiment.

dataset	**name**	N(Mil)	M(Mil)
as-skitter	*sk*	1.70	11.10
youtube	*yt*	3.22	12.22
live-journal	*lj*	4.85	42.85
com-orkut	*orkut*	3.07	117.19
uk-2002	*uk*	18.52	261.79
friendster	*fs*	65.61	1806.07

Algorithms. We implemented and compared seven algorithms:

- Edge: EdgeJoin (Chapter 4.2) with early filtering (Chapter 4.5.3).
- Mul: MultiwayJoin (Chapter 4.2).
- Star: StarJoin (Chapter 4.2) with early filtering (Chapter 4.5.3).
- TTBS: TwinTwigJoin (Chapter 4.3) without optimization.
- TTOA: TTBS + order-aware cost reduction (Chapter 4.5.1).
- TTLB: TTOA + workload skew reduction (Chapter 4.5.2).
- TT: TTLB + early filtering (Chapter 4.5.3).

All algorithms were implemented using Hadoop (version 1.2.1) with Java 1.6. Note that the early filtering strategy (Chapter 4.5.3) was also applied in Edge and Star, and all the optimization strategies introduced in [AFU13] 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 3 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 Figure 4.6 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.

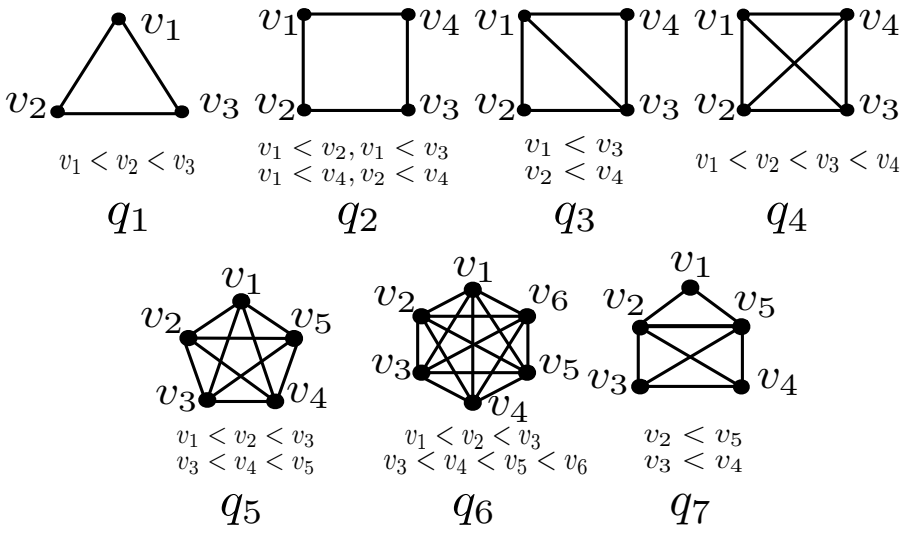
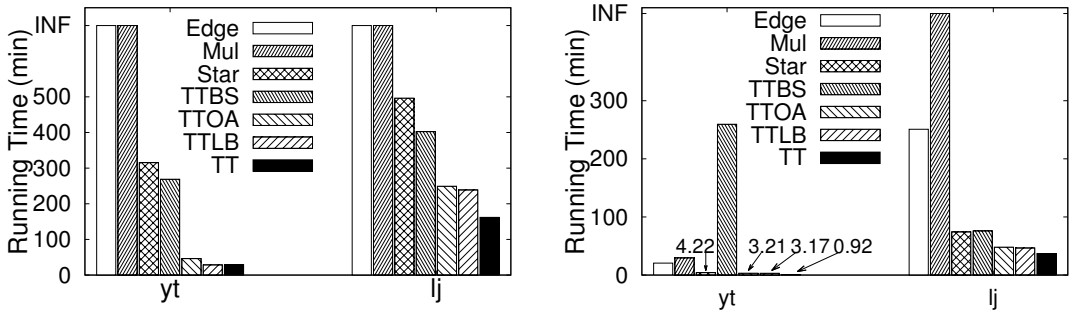


Figure 4.6: Queries used in the TwinTwigJoin experiment.

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 Figure 4.7. We also list the size of the output (see Table 4.2) 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 Chapter 4.3.2, as lower cost generally results in better performance.

As shown in Figure 4.7, 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

Figure 4.7: The results of Exp-1: Vary Algorithms.

Table 4.2: 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 4.2, Edge has to deal with over 9.9

billion data from the third round, yet there are two more rounds to complete the task, in which more partial results are generated.

In most cases, Star is slower than TTBS, which demonstrates the instance optimality of TwinTwig decomposition in Theorem 4.1. 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.

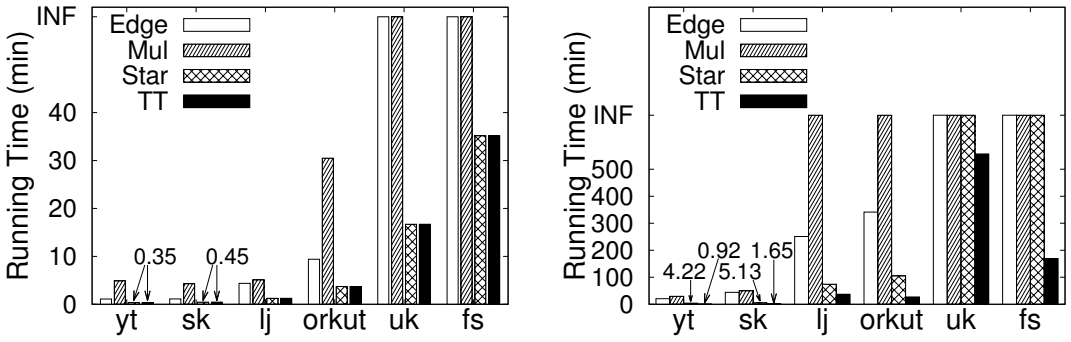
TTOA performs better than TTBS in all cases, which verifies the effectiveness of the order-aware cost reduction strategy, and TTLB outperforms TTOA in all cases, which is consistent with the analysis in Chapter 4.5.2. TT consistently outperforms all other algorithms for all test cases. Comparing TT to TTLB, we observe from Table 4.2 that TTLB generates 10 billion more data than TT, which shows the effectiveness of early filtering. In the rest of the experiments, we exclude the results of TTBS, TTOA, and TTLB, since their relative performances are similar to those shown in Figure 4.7. 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 4.1 and show our results for query q_1 and q_4 for algorithms Edge, Mul, Star, and TT.

Figure 4.8(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*.

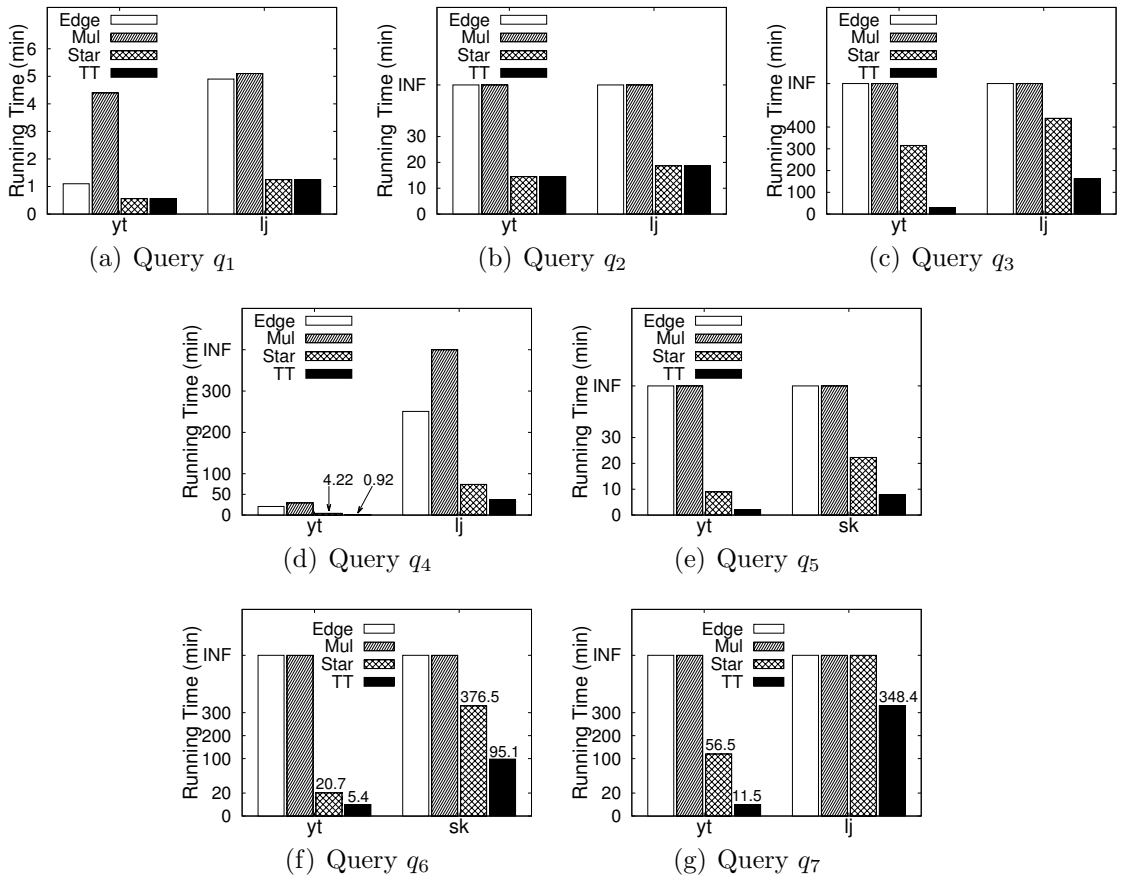
The testing results for q_4 are shown in Figure 4.8(b). TT is 5 times faster than Star on *orkut*, and is only 2 times faster than Star on *lj*. This is because that the larger the average degree of the data graph is, the better performance TT has over Star. The average degree of *orkut*, which is 76, 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.



(a) Query q_1 (b) Query q_4

Figure 4.8: The results of Exp-2: Vary Datasets.

Exp-3: Vary Queries. We evaluated all queries q_1 to q_7 in Figure 4.6. The results are illustrated in Figure 4.9(a) to Figure 4.9(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



(a) Query q_1 (b) Query q_2 (c) Query q_3

(d) Query q_4 (e) Query q_5

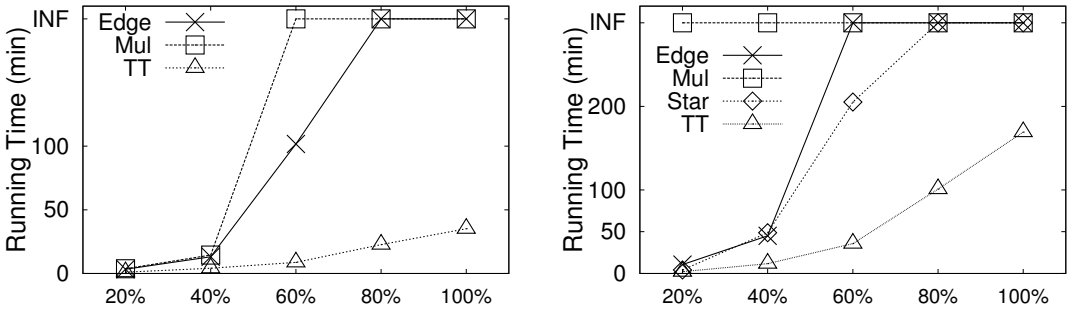
(f) Query q_6 (g) Query q_7

Figure 4.9: The results of Exp-3: Vary Queries.

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 Figure 4.10(a) and Figure 4.10(b) respectively.

We omit the curve of Star in Figure 4.10(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.

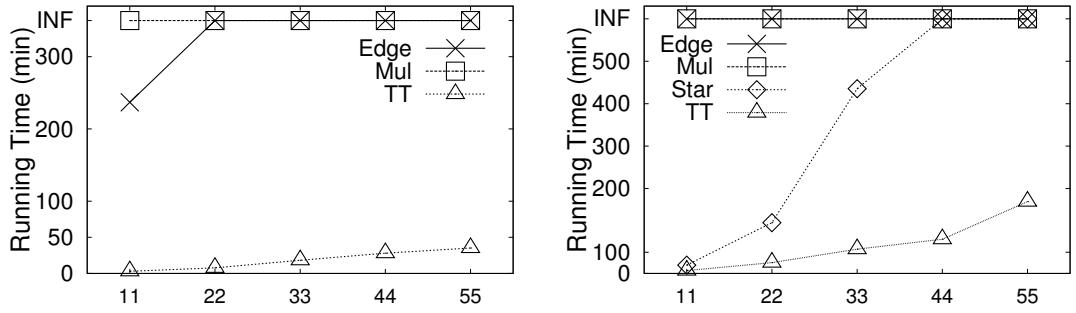


(a) Query q_1 on *fs* (b) Query q_4 on *fs*

Figure 4.10: The results of Exp-4: Vary Graph Size.

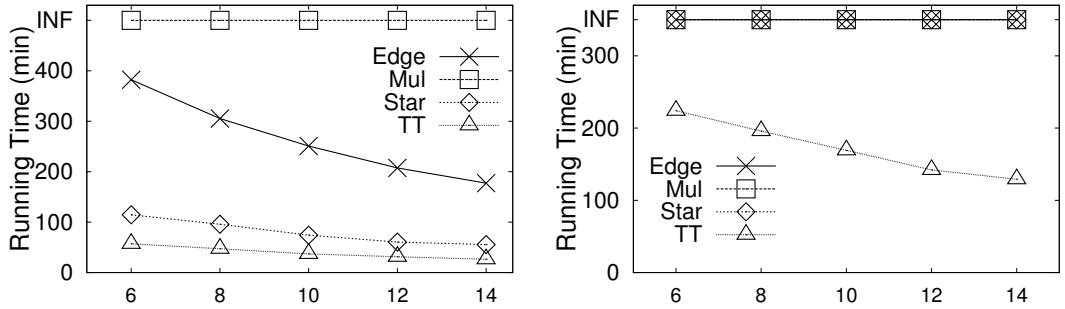
Exp-5: Vary Average Degree. We fixed the set of nodes and randomly sample 20%, 40%, 60%, 80% and 100% edges from the original graph *fs* to generate graphs with average degrees from 11 to 55, and tested the algorithms using queries q_1 and q_4. The results are shown in Figure 4.11(a) and Figure 4.11(b) respectively. We omit the curve of Star in Figure 4.11(a) since Star is the same as TT when evaluating q_1. Edge and Mul fail at the very beginning. In Figure 4.11(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 Chapter 4.3.

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 Figure 6.4(a) and Figure 6.4(b) respectively. As shown in Figure 6.4(a), when the number of slave nodes increases,



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

Figure 4.11: The results of Exp-5: Vary Average Degree



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

Figure 4.12: The results of Exp-6: Vary Slave Nodes

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 Figure 6.4(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 Figure 4.11 thus are omitted due to lack of space.

Exp-7: Small-Degree Assumption. In this experiment, we show that the small-

Table 4.3: 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

degree assumption A_4 (refer Chapter 4.4) 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 Equation 4.2) 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 4.3, we show the value of α when evaluating q_1 and q_4 in the datasets *sk*, *yt* and *lj*, respectively. As we can see, the cost in G^* are actually the dominate part.

4.7 Chapter Conclusion

In this chapter, we proposed the TwinTwigJoin algorithm based on the star-based join framework in MapReduce. We have shown that under reasonable assumptions, TwinTwigJoin is instance optimal in the star join. An A*-based solution was given to compute the optimal join plan. We further improved our approach using three novel optimization strategies and extend our approach to handle the power-law random-graph model. We conducted extensive performance studies on real large graphs with up to billions of edges to demonstrate the effectiveness of our approach.

Chapter 5

SEED: Optimal Graph-based Bushy Join

Although TwinTwigJoin is instance optimal in the star-based join framework, it still suffers from scalability issues due to the constraints of the star join. Motivated by this, we propose the graph-based bushy join framework in this chapter. We first summarize the two constraints that TwinTwigJoin encounters, namely (1) the simple graph storage mechanism that only supports star as the join unit, and (2) the left-deep join structure that may result in sub-optimal solution. We resolve the first constraint by exploring more structures as the join units. In order to do so, we need to involve more edges to each local graph in the basic storage mechanism, but we soon notice that it is storage-inefficient. We carefully consider the tradeoff between the storage efficiency and the availability of join unit, which results in the development of the SEED algorithm that implements the SCP mechanism to support star and clique as the join units. Then we break though the second limit by devising an dynamic-programming algorithm to compute the optimal **bushy join** plan. We also show that it is beneficial to allow overlapping edges among

the join units. Recall that we apply the ER model to estimate the result size for TwinTwigJoin in Chapter 4.3.2. We further propose to use the PR model in the cost analysis for more realistic estimation. We ultimately show that SEED significantly outperforms TwinTwigJoin in the experiment.

Algorithm 4: SubgraphEnum-Graph(data graph G, pattern graph P)

Input : G : The data graph, stored as $\Phi(G) = \{G_u \mid u \in V(G)\}$,
P : The pattern graph.

Output : $R(P)$: All Matches of P in G.

$\mathcal{E}_o \leftarrow \mathsf{OptExecPlan}(G, P)$ (Algorithm 5);
for $i = 1$ **to** t **do**
  $R(P_i) \leftarrow R(P_i^l) \bowtie R(P_i^r)$ according to $\mathcal{E}_o$ (using map^i and reduce^i);
return $R(P_t)$;
function map^i(**key**: $\emptyset$; **value**: Either a match $f \in R(P_i^l)$, $h \in R(P_i^r)$ or $G_u \in \Phi(G)$)
  $V_k = \{v_{k_1}, v_{k_2}, \ldots, v_{k_s}\} \leftarrow V(P_i^l) \cap V(P_i^r)$;
  if P_i^l is a join unit **then** $\mathsf{genJoinUnit}(P_i^l, G_u, V_k)$;
  else output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_s})); f)$;
  if P_i^r is a join unit **then** $\mathsf{genJoinUnit}(P_i^r, G_u, V_k)$;
  else output $((h(v_{k_1}), h(v_{k_2}), \ldots, h(v_{k_s})); h)$;
function $\mathsf{genJoinUnit}(p, G_u, V_k = \{v_{k_1}, v_{k_2}, \ldots, v_{k_s}\})$
  $R_{G_u}(p) \leftarrow$ all matches of p in G_u;
  forall the match $f \in R_{G_u}(p)$ **do**
    output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_s})); f)$;
function reduce^i: The same as Algorithm 1

5.1 Graph-based Join Framework

Constraints of TwinTwigJoin. To summarize, there are two major drawbacks of TwinTwigJoin that inherit from the star join framework. First, the simple graph storage mechanism $\Phi^0(G)$ only supports star as the join unit, which can result in severe performance bottlenecks. Although TwinTwigJoin uses TwinTwig as a substitution, the issues are only mitigated but not evaded, especially when handling nodes with very large degree. In addition, TwinTwigJoin must process at least $\frac{m}{2}$ rounds, which limits its utilization for complex pattern graph. Second, the left-deep join plan may produce sub-optimal solution as it only considers the left-deep searching space [JK84].

We tackle the issues of TwinTwigJoin by introducing the graph-based join framework with more advanced graph storage mechanism and *the optimal bushy join structure*.

SCP **Graph Storage.** We have known that the simple graph storage $\Phi^0(G)$ used in TwinTwigJoin only supports star as the join unit. We say $\Phi(G)$ is χ*-preserved* if $\Phi(G)$ supports χ to be a join unit, and is strictly χ-preserved, if χ is the only available join unit. Clearly $\Phi^0(G)$ is strictly star-preserved. In the exploration of other graph storage mechanisms that support more join units, we specifically define the **S**tar-**C**lique-**P**reserved (SCP) storage mechanism as:

Definition 5.1. *(*SCP ***storage mechanism)*** $\Phi(G) = \{G_u \mid u \in V(G)\}$ *is an* SCP *storage mechanism, if both star and clique (a complete graph) are supported to be the join units by* $\Phi(G)$*, and it is strictly* SCP*, if star and clique are the only supported join units.*

Here clique is particularly considered to embody its good features that can facilitate the theoretical analysis for designing the graph storage mechanism (details

in Chapter 5.2.1). With clique and other structures as alternatives in the SCP mechanism, we can avoid processing star where possible, which not only saves the cost in a single run, but reduces the rounds of execution as a whole.

Example 5.1. *The plans $\mathcal{E}_1$ and $\mathcal{E}_3$ shown in Figure 3.1 are both left-deep joins, but $\mathcal{E}_1$ uses triangles, while $\mathcal{E}_3$ uses* TwinTwig*s as the join units. Intuitively, we expect that $\mathcal{E}_1$ draws smaller cost as a triangle attaches much fewer matches than a two-edge* TwinTwig*, and $\mathcal{E}_1$ completes in three rounds, while $\mathcal{E}_3$ needs four.*

Bushy Join. Here we solve the subgraph enumeration by processing the join in Equation 3.1 using bushy join. Specifically, the following join is processed in the i[-th] round:

$$R(P_i) = R(P_i^l) \bowtie R(P_i^r), \tag{5.1}$$

where P_i^l and P_i^r are called the left and right join patterns regarding P_i, respectively. The left (or right) join pattern can be either a join unit, or a partial pattern processed in an earlier round.

Opposed to the left-deep join where at least one of P_l^i and P_r^i must be a join unit, bushy join is a more general structure without such a constraint. We exploit the bushy join structure to search for the optimal solution in the whole searching space in order to guarantee the optimality.

Algorithm. We show the graph-based join framework in Algorithm 4. Given the pattern graph P, we first compute the optimal execution plan $\mathcal{E}_o$ in line 1 using Algorithm 5 (details in Chapter 5.2.3) [1]. According to the optimal execution plan $\mathcal{E}_o$, the i[-th] join in Equation 5.1 is processed using MapReduce via map^i and

[1]Note that in Algorithm 1, we only need to compute the decomposition which already determines the left-deep join plan. The algorithm to compute the optimal bushy join plan is much more complicated

reduce^i (line 3). We apply the same reduce^i as in Algorithm 1, thus we focus on map^i here.

The function map^i is shown in lines 5-10. The inputs of map^i are either a match $f \in R(P_i^l)$, a match $h \in R(P_i^r)$ or $(u; G_u)$ for all $G_u \in \Phi(G)$ if we are dealing with a join unit (line 5). We first calculate the join key $\{v_{k_1}, v_{k_2}, \ldots, v_{k_s}\}$ using $V(P_i^l) \cap V(P_i^r)$ (line 6) as in Algorithm 1. Then we compute the matches of P_i^l and P_i^r. Take P_i^l for example. We know whether P_i^l is a join unit in current round according to the execution plan. If P_i^l is a join unit, we invoke genJoinUnit (P_i^l, G_u, V_k) (line 7) to compute the matches of P_i^l in G_u for each $G_u \in \Phi(G)$ (lines 12-14). Note that we fully compute $R(P_i^l)$ by merging $R_{G_u}(P_i^l)$ for all $G_u \in \Phi(G)$ according to Definiton 3.1. If P_i^l is not a join unit, the matches of P_i^l must have been computed in previous round. Then we directly fetch the partial results and output them with the join key (line 8).

Note that Algorithm 4 actually generalizes Algorithm 1, where we can consider each P_i^l as P_{i-1}, and P_i^r as p_i, and the genJoinUnit function can be implemented to compute the matches of star in G_u^0. Furthermore, as we mentioned earlier, the algorithm can remove duplication caused by automorphism of the pattern graph using the method in Chapter 4.3.5, and we do not further discuss here.

5.2 SEED Algorithm

On top of the graph-based join framework, we introduce the SEED algorithm, short for **S**ubgraph **E**num**E**ration in **D**istributed context, that implements the SCP graph storage mechanism to support clique and star as the join units (Chapter 5.2.1). We also introduce how to utilize the PR model for cost analysis (Chapter 5.2.2) and how to compute the optimal bushy join plan (Chapter 5.2.3).

5.2.1 Beyond Stars: SCP Graph Storage

In this subchapter, we propose an efficient SCP storage mechanism, in which each local graph introduces a small number of extra edges to the simple local graph in $\Phi^0(G)$. We leverage the PR model for analysis. Denote $\widetilde{w}$ as the second-order average degree, which can be computed as [CLV03a]:

$$\widetilde{w} = (\sum_{i=1}^{N} w_i^2)/(\sum_{i=1}^{N} w_i) = \Psi w^{\beta-2} w_{max}^{3-\beta},$$

where $\Psi = \frac{(\beta-2)^{\beta-1}}{(3-\beta)(\beta-1)^{\beta-2}}$.

As we mentioned earlier, the simple storage mechanism $\Phi^0(G)$ is not an SCP mechanism. In the following, we will explore two SCP mechanisms - $\Phi^1(G)$ and $\Phi^2(G)$, in which the local graphs for a data node u are denoted as G_u^1 and G_u^2, respectively. In order to use extra strutures as the join units, both mechanisms introduce extra edges to each local graph in $\Phi^0(G)$. Denote $\Delta_u^{(i)} = \mathbb{E}[|E(G_u^i)| - |E(G_u^0)|]$ as the expected number of extra edges introduced by $\Phi^i(G)$ to G_u^0 for $i \in \{1, 2\}$, and let $\Delta_{max}^{(i)} = \max_{u \in V(G)}\{\Delta_u^{(i)}\}$.

SCP Graph Storage. Let $\Phi^1(G) = \{G_u^1 \mid u \in V(G)\}$, where $V(G_u^1) = V(G_u^0)$ and $E(G_u^1) = E(G_u^0) \cup \{(u', u'') \mid u', u'' \in \mathcal{N}(u) \wedge (u', u'') \in E(G)\}$. We divide the edges of each G_u^1 into two parts, the *neighbor edges* $E(G_u^0)$, and the *triangle edges* that close triangles with the neighbor edges. Clearly the triangle edges are extra edges introduced by $\Phi^1(G)$. The following lemma shows that $\Phi^1(G)$ is an SCP storage mechanism.

Lemma 5.1. *Given the storage mechanism $\Phi^1(G) = \{G_u^1 \mid u \in V(G)\}$, p is a join unit w.r.t. $\Phi^1(G)$ if p is a star or a clique.*

Proof. Clearly, $\Phi^1(G)$ is star-preserved since each G_u^1 contains G_u^0. Consider that p is a k-clique (a clique with k nodes). We assume that its matches exist in the

data graph, as otherwise it's trivial. Let $V(p) = \{v_0, v_1, v_2 \ldots, v_{k-1}\}$. We prove that $\forall f \in R_G(p)$, $\exists u \in V(G)$ such that $f \in R_{G^1_u}(p)$. According to Definiton 3.1, this sufficiently concludes that p is a join unit. Consider $G^1_{u_0}$ where $u_0 = f(v_0)$ for a given f. Obviously, $(v_0, v_i) \in E(p)$ for any $1 \leq i \leq k-1$, which gives $(u_0, f(v_i)) \in E(G)$, and more specifically, $(u_0, f(v_i)) \in E(G^1_{u_0})$ as they are the neighbour edges of u_0. Furthermore, as $(v_i, v_j) \in E(p)$ for any $i \neq j$, we have $(f(v_i), f(v_j)) \in E(G)$. We know both $f(v_i)$ and $f(v_j)$ are the neighbours of u_0, thus we have $(f(v_i),\ f(v_j)) \in E(G^1_{u_0})$ as the triangle edge. It is immediate that $f \in R_{G^1_{u_0}}(p)$. □

Despite that $\Phi^1(G)$ is an SCP storage mechanism, it can introduce a lot of extra edges to a certain local graph in $\Phi^0(G)$, as shown in the following lemma.

Lemma 5.2. *Given a* PR *graph* $\mathcal{G}$*, and the node* $u_i \in V(\mathcal{G})$*, we have*

$$\Delta^{(1)}_{u_i} = (\Psi^2 w^{\beta-2} N^{2-\beta}) w_i^2, \text{ and}$$
$$\Delta^{(1)}_{max} = \Psi^2 w^{\beta-1} N^{3-\beta}.$$

Proof. We denote t_i as the expected number of triangles associated with u_i. It is easy to see $G^1_{u_i}$ contains w_i neighbor edges and t_i triangle edges by expectation. Thus:

$$\Delta^{(1)}_{u_i} = \mathbb{E}[|G^1_{u_i}|] - E[d(u_i)] = t_i.$$

Recall $\widetilde{w} = \frac{\sum_{i=1}^N w_i^2}{\sum_{i=1}^N w_i}$ is the second-order average degree. When $2 < \beta < 3$, we can compute $\widetilde{w}$ as [CLV03a]:

$$\widetilde{w} = \Psi w^{\beta-2} w_{max}^{3-\beta}, \tag{5.2}$$

where $\Psi = \frac{(\beta-2)^{\beta-1}}{(3-\beta)(\beta-1)^{\beta-2}}$.

For a given node u_i, we will locate u_j and u_k in the data graph to close a

triangle. Following the PR model, we have:

$$\begin{aligned}\Delta_{u_i}^{(1)} = t_i &= \sum_{j=1}^{N}\sum_{k=1}^{N} \rho w_i w_j \times \rho w_i w_k \times \rho w_j w_k \\ &= w_i^2 \rho \widetilde{w}^2 = w_i^2 \frac{\widetilde{w}^2}{N \times w} \quad (\text{by } \rho = \frac{1}{\sum_{i=1}^{N} w_i} = \frac{1}{N \times w}) \\ &= (\Psi^2 w^{\beta-2} N^{2-\beta}) w_i^2 \quad (\text{ by } w_{max} = \sqrt{wN})\end{aligned}$$

We immediately have $\Delta_{max}^{(1)} = \Psi^2 w^{\beta-1} N^{3-\beta}$ by letting $w_i = w_{max}$. □

Lemma 5.2 shows that the number of extra edges introduced by $G_{u_i}^1$ is nearly proportional to w_i^2, which can cause severe workload skew and thus hamper the scalability of the algorithm.

Discussion 5.1. *Given a node v in P, we say v' is its k-hop neighbor if there is a shortest path of length k between v and v'. A graph is called a one-hop graph if there is a node connecting all other nodes, and a multi-hop graph otherwise. Clearly, star and clique are both one-hop graphs. $\Phi^1(G)$ can actually support all one-hop graphs as the join units, but it is storage-inefficient according to Lemma 5.2. There is a tradeoff between the size of each local graph and the availability of the join units. Clearly, the more join units to support, the more edges should be involved in each local graph. Here we carefully consider such a tradeoff.*

- *All one-hop graphs. $\Phi^1(G)$ supports this option, but is already unaffordable according to Lemma 5.2.*
- *Multi-hop graphs. In order to support multi-hop graphs, we need to involve all two-or-more-hop neighbors of u and relevant edges into G_u, which will render an even larger local graph than G_u^1.*

It hence remains to consider a proper subset of the one-hop graphs for the join units. In order to do so, we need to reduce the number of edges taken into each local graph, which inspires the ultimate storage mechanism in this work, $\Phi^2(G)$.

Efficient SCP Graph Storage. Targeting the deficiencies of $\Phi^1(G)$, we consider a more efficient storage mechanism by leveraging the node order (Definiton 2.1). Specifically, we define $\Phi^2(G) = \{G_u^2 \mid u \in V(G)\}$, where $V(G_u^2) = V(G_u^0)$ and $E(G_u^2) = E(G_u^0) \cup \{(u', u'') \,|\, (u', u'') \in E(G) \wedge u \prec u' \wedge u \prec u''\}$. Compared to G_u^1, G_u^2 only includes the triangle edge when u is the minimum node in the triangle. It is clear that $G_u^0 \subseteq G_u^2 \subseteq G_u^1$. Next, we show that $\Phi^2(G)$ is strictly SCP.

Corollary 5.1. *Consider a pattern graph P and its two nodes v_1, v_2, where v_1 are adjacent to all nodes in $V(P) \setminus v_1$. If an order is assigned between v_1 and v_2 using the symmetry-breaking algorithm (see the appendix), then v_2 must be adjacent to all nodes in $V(P) \setminus v_2$.*

Proof. As v_1 and v_2 are assigned an order using symmetry breaking (see the Appendix for details), there must exist an automorphism (a match from P to itself) σ, such that $\sigma(v_1) = v_2$. By Structure-Preservation (Definiton 2.2), v_2 must be adjacent to all nodes except itself in P. □

Lemma 5.3. *Given the storage mechanism $\Phi^2(G) = \{G_u^2 \mid u \in V(G)\}$, p is a join unit w.r.t. $\Phi^2(G)$* ***if and only if*** *p is a star or a clique.*

Proof. (**If**.) Clearly, $\Phi^2(G)$ is star-preserved since each G_u^2 contains G_u^0. We next show a k-clique is a join unit w.r.t. $\Phi^2(G)$. Given a k-clique p where $V(p) = \{v_1, v_2, \ldots, v_k\}$, we apply a full order $v_1 < v_2 < \cdots < v_k$ for symmetry breaking [GK07]. Assume that the match of p exists. We prove that, $\forall f \in R_G(p)$, $\exists u \in V(G)$, such that $f \in R_{G_u^2}(p)$. Given a match f, we let $f(v_1) = u_1$. For any $2 \leq i < j \leq k$, it is clear that $v_1 < v_i < v_j$ and v_1, v_i, v_j close a triangle in p. This suggests $u_1 \prec f(v_i) \prec f(v_j)$ and $u_1, f(v_i), f(v_j)$ close a triangle in G. By the definition of G_u^2, we have:

$$(u_1, f(v_i)), (u_1, f(v_j)) \text{ and } (f(v_i), f(v_j)) \in E(G_{u_1}^2)$$

In other words, $G^2_{u_1}$ involves every edge of the matched instance of p. Hence, $f \in R_{G^2_{u_1}}(p)$.

(**Only If**.) We prove this by contradiction. Let $V(p) = \{v_1, v_2, \ldots, v_n\}$, and one of its match be $f = (u_1, u_2, \ldots, u_n)$. Assume that p is neither a star nor a clique and the match f is preserved in some G^2_u. Given the structure of G^2_u, there must exist a node v_1 (w.l.g.) in p adjacent to all the other nodes of P, and the match must be preserved in $G^2_{u_1}$. There are at least two nodes that have no edge. Let them be v_2, v_3 (w.l.g.), and we assume that $(u_2, u_3) \notin E(G)$, where $u_2 = f(v_2)$ and $u_3 = f(v_3)$. As p is not a star, we must have at least two nodes v_i, v_j such that $(v_i, v_j) \in E(p)$ and $v_i, v_j \neq v_1$. There are two cases: (1) v_2 or v_3 is one of v_i, v_j; (2) neither v_2 nor v_3 is v_i or v_j. We show both cases are impossible. In case 1, let $v_i = v_2$ and $v_j = v_4$ (w.l.g.). Clearly, $(v_2, v_4) \in E(p)$ implies $(u_2, u_4) \in E(G^2_{u_1})$. By the definition of G^2_u, we have $u_1 \prec u_2$ and $u_1 \prec u_4$, and by the order-preservation match, we must have $v_1 < v_2$ and $v_1 < v_4$. According to Corollary 5.1, v_2 must be adjacent to all other nodes in p, this makes a contradiction as v_2 does not connect v_3. In case 2, let $v_i = v_4$ and $v_j = v_5$ (w.l.g.). Similar to case 1, this implies that v_4 connects v_2, which reduces to case 1 that has made a contradiction already. □

The next lemma shows that $\Phi^2(G)$ brings in much less extra edges than $\Phi^1(G)$ does to each local graph in $\Phi^0(G)$.

Lemma 5.4. *Given a* PR *graph* $\mathcal{G}$ *and a node* $u_i \in V(\mathcal{G})$*, we have*

$$\Delta^{(2)}_{u_i} \leq \Delta^{(2)}_{max} \leq \left[(3-\beta)(4-\beta)^{-\frac{4-\beta}{3-\beta}}\right]^2 \Psi^2 w^{\beta-1} N^{3-\beta}.$$

Proof. Let T'_i denote the set of triangles in T_i that have u_i as the minimum node, and $t'_i = |T'_i|$. We have:

$$\Delta^{(2)}_{u_i} = \mathbb{E}[|E(G^2_{u_i})|] - \mathbb{E}[d(u_i)] = t'_i.$$

Consider a triangle (u_i, u_j, u_k) rooted on u_i such that $u_i \prec u_j$ and $u_i \prec u_k$. Recall that we arrange the nodes in $\mathcal{G}$ by non-decreasing order of their degree. We hence have $j \geq i+1$, $k \geq i+1$. Therefore:

$$\Delta_{u_i}^{(2)} = t'_i = \sum_{j,k=i+1}^{N} \rho w_i w_j \times \rho w_i w_k \times \rho w_j w_k$$
$$= \rho w_i^2 \big(\rho \sum_{j=i+1}^{N} w_j^2\big)^2 = \rho w_i^2 \big(\rho \sum_{j=1}^{N} w_j^2 - \rho \sum_{j=1}^{i} w_j^2\big)^2.$$

Note that $w_1, w_2, \ldots, w_i$ is a degree sequence that has w_i as the maximum degree. We then construct another sequence $w'_1, w'_2, \ldots, w'_i$ with $w'_i = w_i$ that has the same power-law distribution as the original sequence. In other words, they have the same β value. Since $i < N$, it is immediate that the frequency of w'_j must be smaller than that of w_j for any $1 \leq j \leq i$. It is easy to find:

$$\rho \sum_{j=1}^{i} w_j^2 \geq \rho \sum_{j=1}^{i} w_j'^2 = \Psi w^{\beta-2} w_i^{3-\beta}.$$

Therefore, we have:

$$\Delta_{u_i}^{(2)} \leq \varphi(w_i) = \Psi^2 w^{2(\beta-2)} \times \rho w_i^2 \big(w_{max}^{3-\beta} - w_i^{3-\beta}\big)^2.$$

Let $\frac{\partial \varphi(w_i)}{\partial w_i} = 0$, we have $w_i = \frac{w_{max}}{(4-\beta)^{1/(3-\beta)}}$, and:

$$\Delta_{max}^{(2)} \leq \Psi^2 w^{2(\beta-2)} \rho \big[\frac{w_{max}}{(4-\beta)^{1/(3-\beta)}}\big]^2 (\frac{3-\beta}{4-\beta})^2 (w_{max}^2)^{3-\beta}$$
$$= \big[(3-\beta)(4-\beta)^{-\frac{4-\beta}{3-\beta}}\big]^2 \Psi^2 w^{\beta-1} N^{3-\beta}$$

□

In Lemma 5.4, we give an upper bound of $\Delta_{u_i}^{(2)}$, while its value is often much smaller. Specifically, $\Delta_{u_i}^{(2)} = 0$ when $d(u_i) = 1$ and $d(u_i) = \max_{u \in V(G)} d(u)$. In general, we show that $\Delta_{max}^{(2)}$ is much smaller than $\Delta_{max}^{(1)}$. In the PR graph, we set $w = 50$, $N = 1{,}000{,}000$ and vary $\beta = 2.1, 2.3, 2.5, 2.7, 2.9$, and then compare

$\Delta^{(1)}_{max}$ and $\Delta^{(2)}_{max}$ in Table 5.1. It is clear that $\Delta^{(2)}_{max} \ll \Delta^{(1)}_{max}$ in all cases. When β increases, observe that $\Delta^{(2)}_{max}$ decreases significantly while $\Delta^{(1)}_{max}$ remains in the same order. In the experiments (Exp-1 in Chapter 5.3), we further compared $\Delta^{(1)}_u$ with $\Delta^{(2)}_u$ for each data node u using synthetic and real datasets, and the experimental results demonstrate that $\Delta^{(2)}_u \ll \Delta^{(1)}_u$ for all data nodes except those with very small degree in all datasets.

Δ_{max}	$\beta = 2.1$	$\beta = 2.3$	$\beta = 2.5$	$\beta = 2.7$	$\beta = 2.9$
$\Delta^{(1)}_{max}$	141,939	195,260	117,851	76,685	141,797
$\Delta^{(2)}_{max}$	7,652	7,652	2,586	710	174

Table 5.1: The number of extra edges introduced by G^1_u and G^2_u.

$\Phi^2(G)$ only supports star and clique to be the join units by Lemma 5.3. In order to support the other one-hop graphs, we must involve more edges to each $G^2_u \in \Phi^2(G)$, which makes it hard to bound the size of each local graph. We take it as a future work to find other storage mechanisms that support more one-hop graphs as the join units, yet still have size-bounded local graphs. In this work, on top of the graph-based join framework, the SEED algorithm implements $\Phi^2(G)$ to only support star and clique as the join units. In the following, we will refer to G^2_u simply as G_u if not otherwise mentioned.

Implementation Details. Given a data graph G, we implement $\Phi^2(G)$ by constructing G_u for each $u \in V(G)$. Specifically, we first aggregate the neighbor edges of each u to G^2_u. Then we apply existing triangle enumeration approaches such as [AFU13] or our TwinTwigJoin algorithm. For each triangle (u_1, u_2, u_3) generated with $u_1 \prec v_2 \prec v_3$, we add (u_2, u_3) to G_{u_1} as the triangle edge. With G_u for each $u \in V(G)$, we can compute the matches of any star or clique using an in-memory algorithm. The overheads of constructing the new graph storage is dominated by triangle enumeration, which are relatively small, as shown in the experiment,

comparing to the performance gains of using clique as the join unit.

5.2.2 Cost Analysis

We follow the cost model in TwinTwigJoin (Chapter 4.3.2) by summarizing the map data $\mathcal{M}$ (the input and output data of the mapper), shuffle data $\mathcal{S}$ (the data transferred from mapper to reducer) and reduce data $\mathcal{R}$ (the input and output data of reducer) in each round of Algorithm 4. Considering that most real-life graphs are power-law graphs, we further contribute to estimate the number of matches of a pattern graph based on the PR model instead of the ER model, and we show that the PR model delivers more realistic estimations.

Note that the cost is more complicated than that in TwinTwigJoin, as we do not know whether P_i^l (or P_i^r) is a join unit or a partial pattern. Consequently, in order to compute the cost, we first consider an arbitrary join $R(P_\beta) = R(p) \bowtie R(P_\alpha)$, where p is a join unit and P_α, P_β are two partial patterns. Let $\mathcal{M}(P)$, $\mathcal{S}(P)$ and $\mathcal{R}(P)$ denote the map data, shuffle data and reduce data regarding a certain pattern P. According to Algorithm 4, we have:

- The mapper handles the partial pattern P_α and the join unit p in different ways. For P_α, the mapper takes the matches $R(P_\alpha)$ as inputs and directly outputs them with the join key. Therefore, $\mathcal{M}(P_\alpha) = 2R(P_\alpha)$. As for the join unit p, the mapper first reads G_u for each data node u to compute $R(p)$, then outputs the results. Denote $\Delta(G)$ as the set of triangles in G, and it is clear that $\sum_{u \in V(G)} E(G_u) = \Delta(G)$. Therefore, $\mathcal{M}(p) = \Delta(G) + R(p)$.

- The shuffle transfers the mapper's outputs to the corresponding reducer. Therefore, the shuffle data is also the mapper's output data, and we have $\mathcal{S}(P_\alpha) = R(P_\alpha)$ and $\mathcal{S}(p) = R(p)$.

- The reducer takes $R(P_\alpha)$ and $R(p)$ as inputs to compute $R(P_\beta)$. Apparently, the input data are $\mathcal{R}(P_\alpha) = R(P_\alpha)$ and $\mathcal{R}(p) = R(p)$, and the output data are $\mathcal{R}(P_\beta) = R(P_\beta)$.

Summarizing the above, the cost for processing the join unit p in a certain join is:

$$\mathcal{T}(p) = |\mathcal{M}(p)| + |\mathcal{S}(p)| + |\mathcal{R}(p)| = |\Delta(G)| + 3|R(p)| \tag{5.3}$$

and the cost for processing the partial pattern P_α is:

$$\mathcal{T}(P_\alpha) = |\mathcal{M}(P_\alpha)| + |\mathcal{S}(P_\alpha)| + |\mathcal{R}(P_\alpha)| = 5|R(P_\alpha)| \tag{5.4}$$

Note that $R(P_\alpha)$ must have been generated in earlier round, while the cost to output $R(P_\alpha)$ is involved in $\mathcal{T}(P_\alpha)$ for consistency, and $R(P_\beta)$ will be accordingly computed in $\mathcal{T}(P_\beta)$.

Given an execution plan $\mathcal{E} = (D, J)$, where $D = \{p_0, p_1, \ldots, p_t\}$, it is processed using t rounds of joins, and in the i[-th] round the partial results $R(P_i)$ are generated. We compute the cost by putting all costs of processing p_i and P_i together as:

$$\mathcal{C}(\mathcal{E}) = \sum_{i=0}^{t} \mathcal{T}(p_i) + \sum_{i=1}^{t-1} \mathcal{T}(P_i), \tag{5.5}$$

where $\mathcal{T}(p_i)$ and $\mathcal{T}(P_i)$ are computed via Equation 5.3 and Equation 5.4, respectively.

Result-Size Estimation. We need to estimate $|R(P)|$ for a certain P in Equation 5.3 and Equation 5.4. It is obvious that all partial patterns in Algorithm 4 are connected. Given a connected pattern graph P, we next show how to estimate $|R_\mathcal{G}(P)|$ in the PR graph $\mathcal{G}$.

Suppose P is constructed from an edge by extending one edge step by step, and $P^{(1)}$ and $P^{(2)}$ are two consecutive patterns obtained along the process. More specifically, given $v \in V(P^{(1)})$ and $v' \in V(P^{(2)})$ where $(v, v') \notin E(P^{(1)})$, $P^{(2)}$ is

obtained by adding the edge (v, v') to $P^{(1)}$. We let δ and δ' be the degrees of v and v' in $P^{(1)}$, respectively. Here, if $v' \notin V(P^{(1)})$, $\delta' = 0$. Given a match f of $P^{(1)}$, we let $f(v) = u$. We then extend f to generate the match f' of $P^{(2)}$ by locating another node $u' \in V(G)$ where $(u, u') \in E(G)$ and $f'(v') = u'$. Suppose there are by expectation γ matches of $P^{(2)}$ that can be extended from one certain match of $P^{(1)}$, we have:

$$|R_{\mathcal{G}}(P^{(2)})| = \gamma |R_{\mathcal{G}}(P^{(1)})|$$

The value of γ depends on how the edge is extended from $P^{(1)}$ to form $P^{(2)}$. There are two cases, namely, $v' \notin V(P^{(1)})$ and $v' \in V(P^{(1)})$, which are respectively discussed in the following.

(Case 1) $v' \notin V(P^{(1)})$. In this case, a new node v' is introduced to extend the edge (v, v'). The potential match of v', namely u', can be any data node in G. Therefore, we have:

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

where ϕ_i is the probability that u appears as u_i in the matches of $P^{(1)}$.

For ease of analysis, we focus on the relationships between u and its neighbors. Denote k-star(u) as the star with k leaves rooted on u. Clearly, u and its neighbors in the match form a δ-star(u) (Note that the degree of u in the match is equal to

the degree of v in $P^{(1)}$, that is δ). Thus, we have:

$$\begin{aligned}\phi_i &= \mathsf{Pr}(u = u_i \mid u \text{ and its neighbors form a } \delta\text{-star}(u)) \\ &= \frac{\mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u) \mid u = u_i)\mathsf{Pr}(u = u_i)}{\mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u))} \\ &= \frac{\mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u) \mid u = u_i)}{\sum_{j=1}^{N} \mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u) \mid u = u_j)} \\ &= \frac{\mathsf{Pr}(\mathsf{E}_i)}{\sum_{j=1}^{N} \mathsf{Pr}(\mathsf{E}_j)}.\end{aligned}$$

where E_i denotes the event that u and its neighbors form a δ-star(u) given $u = u_i$. Given any node set $\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \subset V(G)$, E_i can be witnessed as there is an edge connecting u_i and u_{t_s} for any $1 \leq s \leq \delta$. According to the PR model, the probability that any u_i and u_j are connected is $\mathsf{Pr}_{i,j} = w_i w_j \rho$, hence we have:

$$\begin{aligned}\mathsf{Pr}(\mathsf{E}_i) &= \sum_{\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \subset V(G)} \mathsf{Pr}_{i,t_1} \times \cdots \times \mathsf{Pr}_{i,t_\delta} \\ &= \sum_{\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \subset V(G)} w_i w_{t_1} \rho \times \cdots \times w_i w_{t_\delta} \rho \\ &= (w_i)^\delta \times C,\end{aligned}$$

where $C = \rho^\delta \sum_{\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \in V(G)} \prod_{s=1}^{\delta} w_{t_s}$.

Note that C is a constant for any i. Therefore:

$$\phi_i = \frac{\mathsf{Pr}(\mathsf{E}_i)}{\sum_{j=1}^{N} \mathsf{Pr}(\mathsf{E}_j)} = \frac{w_i^\delta}{\sum_{j=1}^{N} w_j^\delta}, \tag{5.6}$$

and

$$\gamma = \sum_{i=1}^{N} \frac{w_i^\delta}{\sum_{j=1}^{N} w_j^\delta} w_i = \frac{\sum_{i=1}^{N} w_i^{\delta+1}}{\sum_{i=1}^{N} w_i^\delta}. \tag{5.7}$$

(Case 2) $v' \in V(P^{(1)})$. In this case, a new edge is added between two existing nodes in $P^{(1)}$. In this case, a new edge is added between two existing nodes in $P^{(1)}$. Consider the two nodes v and v' in $P^{(1)}$, and u and u' are their matches in

an arbitrary match of $P^{(1)}$. We compute γ as:

$$\gamma = \mathbb{E}[d(u)d(u')\rho] = \rho\mathbb{E}[d(u)d(u')].$$

We still consider the neighbors of u and u' in the match. Suppose u and u' has δ and δ' neighbors respectively. Denote $\phi_{i,j}$ as the probability that u and u' appear as u_i and u_j in the match. Then:

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

There are two cases. If u and u' have no common neighbor in the match, it is obvious that:

$$\phi_{i,j} = \phi_i\phi_j = \frac{w_i^{\delta}}{\sum_{s=1}^{N} w_s^{\delta}} \frac{w_j^{\delta'}}{\sum_{s=1}^{N} w_s^{\delta'}},$$

where ϕ_i and ϕ_j are computed according to Equation 5.6.

If their neighbors coincide, u and u' are not independent. Let $V_c = \{u_{c_1}, u_{c_2}, \ldots, u_{c_t}\}$ denote the common neighbors of u and u', $V_d = \{u_{d_1}, \ldots, u_{d_{\delta-t}}\}$ denote only u's neighbors and $V'_d = \{u_{d'_1}, \ldots, u_{d'_{\delta'-t}}\}$ denote only u''s neighbors. The structure formed by u, u' and their neighbors is called a *twin star*, and u and u' are the roots of the twin star. We have:

$$\begin{aligned}
\phi_{i,j} &= \mathsf{Pr}(u = u_i, u' = u_j \mid u, u' \text{ root a twin star}) \\
&= \frac{\mathsf{Pr}(u, u' \text{ root a twin star} \mid u = u_i, u' = u_j)\mathsf{Pr}(u = u_i, u' = u_j)}{\mathsf{Pr}(u, u' \text{ root a twin star})} \\
&= \frac{\mathsf{Pr}(\mathsf{E}_{i,j})}{\sum_{s,t=1}^{N} \mathsf{Pr}(\mathsf{E}_{s,t})},
\end{aligned}$$

where $\mathsf{E}_{i,j}$ denotes the event that u and u' root a twin star given $u = u_i$ and $u' = u_j$.

Following the same idea in deriving Equation 5.7, we have:

$$\begin{aligned}\mathsf{Pr}(\mathsf{E}_{i,j}) = &\sum_{V_c \subset V(G)} \sum_{V_d \subset V(G)} \sum_{V'_d \subset V(G)} \mathsf{Pr}_{i,c_1} \times \mathsf{Pr}_{i,c_2} \times \cdots \mathsf{Pr}_{i,c_t} \\ &\times \mathsf{Pr}_{i,d_1} \times \mathsf{Pr}_{i,d_2} \times \cdots \mathsf{Pr}_{i,d_{\delta-t}} \times \mathsf{Pr}_{j,c_1} \times \mathsf{Pr}_{j,c_2} \times \cdots \mathsf{Pr}_{j,c_t} \\ &\times \mathsf{Pr}_{j,d'_1} \times \mathsf{Pr}_{j,d'_2} \times \cdots \mathsf{Pr}_{j,d'_{\delta'-t}} \\ &= w_i^{\delta} w_j^{\delta'} \times \sum_{V_c \subset V(G)} \prod_{s=1}^{t} \rho^2 w_{c_s}^2 \times \sum_{V_d \subset V(G)} \prod_{s=1}^{\delta-t} \rho w_{d_s} \\ &\times \sum_{V'_d \subset V(G)} \prod_{s=1}^{\delta'-t} \rho w_{d'_s} = w_i^{\delta} w_j^{\delta'} \times C,\end{aligned} \tag{5.8}$$

where

$$\begin{aligned}C = &\sum_{V_c \subset V(G)} \prod_{s=1}^{t} \rho^2 w_{c_s}^2 \times \sum_{V_d \subset V(G)} \prod_{s=1}^{\delta-t} \rho w_{d_s} \\ &\times \sum_{V'_d \subset V(G)} \prod_{s=1}^{\delta'-t} \rho w_{d'_s}.\end{aligned}$$

Note that C is a constant for any i, j. Therefore:

$$\begin{aligned}\phi_{i,j} &= \frac{\mathsf{Pr}(\mathsf{E}_{i,j})}{\sum_{s,t=1}^{N} \mathsf{Pr}(\mathsf{E}_{s,t})} = \frac{w_i^{\delta} w_j^{\delta'}}{\sum_{s,t=1}^{N} w_s^{\delta} w_t^{\delta'}} \\ &= \frac{w_i^{\delta}}{\sum_{s=1}^{N} w_s^{\delta}} \frac{w_j^{\delta'}}{\sum_{s=1}^{N} w_s^{\delta'}}.\end{aligned}$$

As a result of both cases, we have:

$$\begin{aligned}\gamma &= \rho \sum_{i,j=1}^{N} \phi_{i,j} w_i w_j \\ &= \rho \times \frac{\Sigma_{i=1}^{N} w_i^{\delta+1}}{\Sigma_{i=1}^{N} w_i^{\delta}} \times \frac{\Sigma_{j=1}^{N} w_j^{\delta'+1}}{\Sigma_{j=1}^{N} w_j^{\delta'}}.\end{aligned} \tag{5.9}$$

Given Equation 5.7 and Equation 5.9, we compute $|R_{\mathcal{G}}(P)|$ for any connected pattern graph P as follows. First, we run Depth-First-Search (DFS) over P to obtain the DFS-tree. Then, starting from an edge e with $|R_{\mathcal{G}}(e)| = 2M$, we apply Equation 5.7 iteratively to compute the size of the tree. Finally, we apply Equation 5.9 iteratively as we extend the non-tree edges. Note that, given a graph G,

the γ calculated by Equation 5.7 or Equation 5.9 only depends on δ and δ', thus can be precomputed.

$\lvert R_{\mathcal{G}}(P)\rvert$	$\beta = 2.1$	$\beta = 2.3$	$\beta = 2.5$	$\beta = 2.7$	$\beta = 2.9$
$\lvert R_{\mathcal{G}}(P_2^{ld})\rvert$	67618.5	14632.5	1993.0	610.2	83.4
$\lvert R_{\mathcal{G}}(P_2^{b})\rvert$	230.2	69.5	16.3	6.2	1.5

Table 5.2: The number of the matches of P_2^{ld} and P_2^b in the PR graph (in billions).

Remark 5.1. *The plans $\mathcal{E}_1$ and $\mathcal{E}_2$ shown in Figure 3.1 are actually the optimal execution plans computed using the* ER *model and the* PR *model, respectively. Observe that $\mathcal{E}_1$ differs from the $\mathcal{E}_2$ in the second round where P_2^{ld} is processed instead of P_2^b. Generally, we have $\lvert R(P_2^{ld})\rvert < \lvert R(P_2^b)\rvert$ in the* ER *model [LQLC15], but $\lvert R(P_2^{ld})\rvert \gg \lvert R(P_2^b)\rvert$ in the* PR *model. As a result, $\mathcal{E}_1$ and $\mathcal{E}_2$ are returned as the optimal plan regarding the* ER *model and* PR *model, respectively. Next we consider an* ER *graph $\Re$ and a* PR *graph $\mathcal{G}$ with $N = 1{,}000{,}000$ and $M = 25{,}000{,}000$, and compute $\lvert R(P_2^{ld})\rvert$ and $\lvert R(P_2^b)\rvert$ in both graphs for a comparison. According to [LQLC15], we have $\lvert R_{\Re}(P_2^{ld})\rvert = 0.78$, and $\lvert R_{\Re}(P_2^b)\rvert = 312$. Then we compute $\lvert R_{\mathcal{G}}(P_2^{ld})\rvert$ and $\lvert R_{\mathcal{G}}(P_2^b)\rvert$ using the proposed method, and show the results with various power-law exponents in Table 5.2. It is clear to see that $\lvert R_{\mathcal{G}}(P_2^{ld})\rvert \gg \lvert R_{\mathcal{G}}(P_2^b)\rvert$ in all cases. In Chapter 5.3, we further experimented using real-life graphs, which confirms that the* PR *model offers more realistic estimation and consequently renders better execution plan.*

5.2.3 Optimal Execution Plan

In this subchapter, we propose a dynamic-programming algorithm to compute the optimal bushy join plan for SEED. We further consider overlaps among the join units.

Non-overlapped Case. To show the basic idea, we first introduce the non-overlapped case.

Definition 5.2. *(**Partial Execution**) A partial execution, denoted $\mathcal{E}_{P_\alpha}$, is an execution plan that computes the partial pattern $P_\alpha \subseteq P$.*

Given a partial pattern $P_\alpha \subseteq P$, the optimal partial execution plan of P_α satisfies:

$$\mathcal{C}(\mathcal{E}_{P_\alpha}) = \begin{cases} 0, \ P_\alpha \text{ is a join unit,} \\ \min\limits_{P^l_\alpha \subset P_\alpha} \{\mathcal{C}(\mathcal{E}_{P^l_\alpha}) + \mathcal{T}(P^l_\alpha) + \mathcal{C}(\mathcal{E}_{P^r_\alpha}) + \mathcal{T}(P^r_\alpha)\}, \text{ otherwise.} \end{cases} \tag{5.10}$$

where $P^r_\alpha = P_\alpha \setminus P^l_\alpha$, $\mathcal{T}(P^l_\alpha)$ and $\mathcal{T}(P^r_\alpha)$ are computed via Equation 5.3 or Equation 5.4 depending on whether they are join units or partial patterns. The optimal partial execution $\mathcal{E}_{P_\alpha}$ is obtained while minimizing the sum of the cost of the optimal $\mathcal{E}_{P'_l}$ and $\mathcal{E}_{P'_r}$, and the cost of processing the join $R(P_\alpha) = R(P^l_\alpha) \bowtie R(P^r_\alpha)$, that is $\mathcal{T}(P^l_\alpha) + \mathcal{T}(P^r_\alpha)$. Note that $\mathcal{C}(\mathcal{E}_{P_\alpha}) = 0$ if P_α is a join unit, as no join is needed to compute the results of a join unit (Definiton 3.1).

We use a hash map $\mathcal{H}$ to maintain the so far best partial execution for each $P_\alpha \subseteq P$. The entry of the hash map for P_α has the form $(P_\alpha, \mathcal{T}, \mathcal{C}, P^l_\alpha, P^r_\alpha)$, where $\mathcal{T}$ is an auxiliary cost computed via either Equation 5.3 or Equation 5.4, $\mathcal{C}$ is the so far best cost $\mathcal{C}(E_{P_\alpha})$ while evaluating P_α, P^l_α is the left-join pattern when the current best cost is obtained, and P^r_α is the corresponding right-join pattern, where $P^r_\alpha = P_\alpha \setminus P^l_\alpha$, as no overlap is considered. The hash map is indexed by P_α and we can access one specific item I for a certain P_α via $\mathcal{H}_{P_\alpha}(I)$, where $I \in \{\mathcal{T}, \mathcal{C}, P^l_\alpha, P^r_\alpha\}$.

The algorithm to compute the optimal execution plan is shown in Algorithm 5. In line 5, We initialize an entry in the hash map for each connected $P_\alpha \subseteq P$ that is potentially a partial pattern (line 2). Note that we precompute $\mathcal{T}(P_\alpha)$ for each P_α. To find the optimal execution plan for P, we need to accordingly find the

Algorithm 5: OptExecPlan(data graph G, pattern graph P)

Input : The data graph G and the pattern graph P

Output : The optimal execution plan w.r.t. P.

```
1  forall the P_α ⊆ P, s.t. P_α is connected do
2      H ← H ∪ (P_α, T(P_α), ∞, ∅, ∅);
3  for s = 1...m, where m = |E(P)| do
4      forall the P_α ⊂ P s.t. P_α is connected and |E(P_α)| = s do
5          if P_α is a join unit then
6              H_{P_α}(C) = 0;
7          else
8              forall the P_α^l ⊂ P_α s.t. P_α^l and P_α^r = P_α \ P_α^l are connected do
9                  C ← H_{P_α^l}(C) + H_{P_α^l}(T) + H_{P_α^r}(C) + H_{P_α^r}(T);  if C < H_{P_α}(C) then
10                     H_{P_α}(C) ← C;
11                     H_{P_α}(P_α^l) ← P_α^l;  H_{P_α}(P_α^r) ← P_α^r;
12 E_o ← ComputePlan(H, P);
13 return E_o;
```

optimal partial execution plans for all P's subgraphs, in non-decreasing order of their sizes. The algorithm performs three nested loops. The first loop in line 3 confines the size of the partial patterns to s, and the second loop enumerates all possible partial patterns with size s (line 4). If the current partial pattern P_α is a join unit, we simply set the corresponding cost to 0 (line 6). Otherwise, the third loop is triggered to update the optimal execution plan for P_α (line 8). We enumerate all P_α^l (and $P_\alpha^r = P_\alpha \setminus P_\alpha^l$), and for each P_α^l where P_α^l and P_α^r are both connected, we compute $\mathcal{C}(\mathcal{E}_{P_\alpha})$ via Equation 5.10 (line 9). In this way, we finally find the P_α^l to minimize $\mathcal{C}(\mathcal{E}_{P_\alpha})$, and update the entry of P_α by setting $\mathcal{H}_{P_\alpha}(\mathcal{C})$,

$\mathcal{H}_{P_\alpha}(P_\alpha^l)$ and $\mathcal{H}_{P_\alpha}(P_\alpha^r)$ correspondingly (line 10-11). After all the entries in the hash map are computed, we first look up the entry for P to locate the P_α^l and P_α^r and repeat the procedure recursively on P_α^l and P_α^r until $P_\alpha^l = \emptyset$. In this way, we compute the optimal execution plan (line 12).

Lemma 5.5. *The space complexity and time complexity of Algorithm 5 are $O(2^m)$ and $O(3^m)$, respectively.*

Proof. We first show the space complexity. Each entry in $\mathcal{H}$ is uniquely identified by the partial pattern P_α, and there are at most 2^m partial patterns, which consumes $O(2^m)$ space.

Next we prove the time complexity. There are at most $\binom{m}{s}$ partial patterns of P sized s, which trigger $\binom{m}{s}$ calls over the second loop. For each partial pattern P_α of size s, we enumerate all possible connected subgraphs of it as P_α^l, which incurs at most 2^s calls. By pre-computing any connected partial patterns of P (which use $O(2^m)$ space and time), we can verify the connectedness of P_α^l and P_α^r in $O(1)$ time. Moreover, updating the entry has the cost $O(1)$. Note that $\sum_{s=1}^{m} \binom{m}{s} \cdot 2^s = (1+2)^m = 3^m$. Therefore, the time complexity of Algorithm 5 is $O(3^m)$. □

Discussion 5.2. *In practice, the processing time for Algorithm 5 is much smaller than $O(3^m)$ since we require that all partial patterns are connected.*

Overlapped Case. The following lemma inspires us to consider overlaps among the join units.

Lemma 5.6. *Given a pattern graph P, and another pattern graph P^-, where $v, v' \in V(P^-)$, $(v, v') \notin E(P^-)$ and $P = P^- \cup \{(v, v')\}$, we have:*

$$|R(P)| \leq |R(P^-)|.$$

Proof. Apparently, $\forall f \in R(P), f \in R(P^-)$. Therefore, $|R(P)| \leq |R(P^-)|$. □

Example 5.2. *We have actually shown overlaps among the join units in Figure 3.1. For example, we have* $E(p_0) \cap E(p_1) = \{(v_1, v_3)\}$ *in the bushy tree* $\mathcal{E}_2$. *Let* $p_1^- = p_1 \setminus (v_1, v_3)$. *In the non-overlapped case, we will execute* $R(P_1') = R(p_0) \bowtie R(p_1^-)$ *instead. Clearly,* $|R(p_1)| \leq |R(p_1^-)|$ *according to Lemma 5.6, and hence the plan with overlaps is better.*

A naive solution to allow the join units to overlap and still guarantee the optimality in Algorithm 5 is: when we evaluate $P_\alpha^r = P_\alpha \setminus P_\alpha^l$ in line 8, we further enumerate all possible P_α^{r*}, where P_α^{r*} are all *connected* structures formed by adding any subset of $E(P_\alpha^l)$ to P_α^r. As a result, the time complexity is of the order:

$$\sum_{s=1}^{m} \binom{m}{s} \cdot \sum_{t=1}^{s} \binom{s}{t} 2^t = 4^m.$$

All or Nothing. The time complexity of computing the optimal execution in the overlapped case can be reduced to $O(3^m)$ with some practical relaxation. Given a partial pattern P_α, and its left-join (resp. right-join) pattern P_α^l (resp. P_α^r) (P_l' and P_α^r may overlap), we define the redundant node as:

Definition 5.3. ***(Redundant Node)*** *A node* $v_r \in V(P_\alpha^l) \cap V(P_\alpha^r)$ *is a redundant node w.r.t.* $P_\alpha = P_\alpha^l \cup P_\alpha^r$, *if* $P_\alpha = (P_\alpha^l \setminus v_r) \cup P_\alpha^r$ *or* $P_\alpha = P_\alpha^l \cup (P_\alpha^r \setminus v_r)$.

In other words, the removal of a redundant node from either P_α^l or P_α^r does not affect the join results. Denote V_r as a set of redundant nodes w.r.t. $P_\alpha = P_\alpha^l \cup P_\alpha^r$. We further define the cut nodes V_c and the cut edges E_c as follows:

$$V_c(P_\alpha^l, P_\alpha^r) = \big(V(P_\alpha^l) \cap V(P_\alpha^r)\big) \setminus V_r$$

$$E_c(P_\alpha^l, P_\alpha^r) = \{(v, v') \mid (v, v') \in E(P_\alpha) \wedge v, v' \in V_c(P_\alpha^l, P_\alpha^r)\}.$$

Example 5.3. *In Figure 5.1, we show a partial pattern P_α and its left-join (resp. right-join) pattern P_α^l (resp. P_α^r). Clearly, v_4 is a redundant node since $P_\alpha = P_\alpha^l \cup (P_\alpha^r \setminus v_4)$, and we have $V_c(P_\alpha^l, P_\alpha^r) = \{v_2, v_3\}$ and $E_c(P_\alpha^l, P_\alpha^r) = \{(v_2, v_3)\}$.*

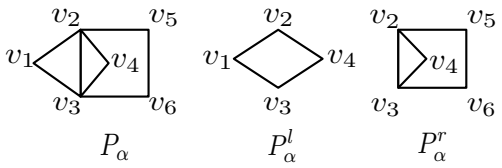


Figure 5.1: The redundant node, cut nodes and cut edges.

Based on the cut edges, we introduce an *all-or-nothing* strategy, which reduces the time complexity of computing the optimal execution plan with overlaps to $O(3^m)$. Specifically, when we evaluate $P_\alpha^r = P_\alpha \setminus P_\alpha^l$ in Algorithm 5 (line 8), instead of enumerating P_α^{r*} by considering all subsets of $E(P_\alpha^l)$ in the naive solution, we only consider adding all the cut edges w.r.t. $P_\alpha = P_\alpha^l \cup P_\alpha^r$, or none of them. We show that the all-or-nothing strategy returns an execution plan that is almost as good as the naive solution.

Denote $\mathcal{E}_o = \{D_o, J_o\}$ as the optimal execution plan with overlaps obtained by the naive solution and $\mathcal{E}_o'$ as the best execution plan obtained by the all-or-nothing strategy. We know in the i[-th] round of the execution plan $\mathcal{E}_o$, the following join is processed:

$$R(P_i) = R(P_i^l) \bowtie R(P_i^r), \forall 1 \leq i \leq |D_o| - 1.$$

We then construct an intermediate execution plan $\widetilde{\mathcal{E}}$ by replacing each of the above join as $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$, where $\widetilde{P_i^r} = P_i^r \cup \{e_1, e_2, \ldots, e_s\}$, and each $e_i \in E_c(P_i^l, P_i^r) \wedge e_i \notin E(P_i^r)$. In other words, the alternative right-join pattern $\widetilde{P_i^r}$ is obtained by adding all the cut edges to P_i^r. It is trivial when $\widetilde{P_i^r} = P_i^r$. Otherwise, we first generate $R(\widetilde{P_i^r})$ by performing the joins $R(P_i^r) \bowtie R(e_1) \bowtie \cdots \bowtie R(e_r)$ sequentially, and each join handles a cut edge. Then we execute the join $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$.

Leveraging the intermediate execution plan $\widetilde{\mathcal{E}}$, we prove that $\mathcal{E}'_o$ (the best execution plan computed by "all-or-nothing" strategy) has the cost of the same order as $\mathcal{E}_o$ (the optimal solution). We first prove $\mathcal{C}(\widetilde{\mathcal{E}}) = \Theta(\mathcal{C}(\mathcal{E}_o))$.

Lemma 5.7. *If* $\mathcal{C}(\mathcal{E}_o) \geq \Theta(M)$, *then* $\mathcal{C}(\widetilde{\mathcal{E}}) = \Theta(\mathcal{C}(\mathcal{E}_o))$.

Proof. We divide $\widetilde{\mathcal{E}}$ into two parts. For $1 \leq i \leq t$, the first part, denoted as $\widetilde{\mathcal{E}}_1$, performs the join $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$; the second part, denoted as $\widetilde{\mathcal{E}}_2$, handles the generation of each $R(\widetilde{P_i^r})$ by sequentially joining the matches of each cut edge to $R(P_i^r)$. According to Lemma 5.6, we have:

$$|R(\widetilde{P_i^r})| \leq |R(P_i^r)|. \tag{5.11}$$

Clearly, $\mathcal{C}(\widetilde{\mathcal{E}}) = \mathcal{C}(\widetilde{\mathcal{E}}_1) + \mathcal{C}(\widetilde{\mathcal{E}}_2)$. We accordingly divide the proof into two parts.

(**Part 1**) We prove $\mathcal{C}(\widetilde{\mathcal{E}}_1) \leq \Theta(\mathcal{C}(\mathcal{E}_o))$. Denote $\mathcal{E}_{P_i}$ and $\widetilde{\mathcal{E}}_{P_i}$ as the partial execution of generating $R(P_i)$ in $\mathcal{E}_o$ and $\widetilde{\mathcal{E}}_1$. According to Equation 5.10, we have:

$$\mathcal{C}(\mathcal{E}_{P_i}) = \mathcal{C}(\mathcal{E}_{P_i^l}) + \mathcal{T}(P_i^l) + \mathcal{C}(\mathcal{E}_{P_i^r}) + \mathcal{T}(P_i^r),$$
$$\mathcal{C}(\widetilde{\mathcal{E}}_{P_i}) = \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^l}) + \mathcal{T}(P_i^l) + \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^r}) + \mathcal{T}(\widetilde{P_i^r}),$$

where $\mathcal{T}(P_\alpha)$ are computed via Equation 5.3 or Equation 5.4 depending on whether P_α is a join unit or a partial pattern.

Let $t = |D_o| - 1$ denote the number of rounds of $\mathcal{E}_o$. We prove Part 1 by inducing on $i = 1, 2, \ldots, t$.

When $i = 1$, P_1^l and P_1^r must be the join units, thus $\mathcal{C}(\mathcal{E}_{P_i^l}) = \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^l}) = \mathcal{C}(\mathcal{E}_{P_i^r}) = \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^r}) = 0$. Further, we have $\mathcal{T}(\widetilde{P_1^r}) \leq \mathcal{T}(P_1^r)$ given that $|R(\widetilde{P_1^r})| \leq |R(P_1^r)|$ by Equation 5.11. Hence, $\mathcal{C}(\widetilde{\mathcal{E}}_{P_1}) \leq \Theta(\mathcal{C}(\mathcal{E}_{P_1}))$.

Assume that $\mathcal{C}(\widetilde{\mathcal{E}}_{P_i}) \leq \Theta(\mathcal{C}(\mathcal{E}_{P_i}))$ holds for all $1 < i \leq s-1, s < t$. Consider $i = s$. Note that P_s^l and P_s^r are some P_j with $j < i$, we hence have $\mathcal{C}(\widetilde{\mathcal{E}}_{P_s^l}) \leq \Theta(\mathcal{E}_{P_s^l})$ and $\mathcal{C}(\widetilde{\mathcal{E}}_{P_s^r}) \leq \Theta(\mathcal{E}_{P_s^r})$ by the assumption. Additionally, $\mathcal{T}(\widetilde{P_s^r}) \leq \mathcal{T}(P_s^r)$ by

Equation 5.11. It is immediate that $\mathcal{C}(\widetilde{\mathcal{E}}_{P_s}) \leq \Theta(\mathcal{C}(\mathcal{E}_{P_s}))$. By induction, we have:

$$\mathcal{C}(\widetilde{\mathcal{E}}_1)(= \mathcal{C}(\widetilde{\mathcal{E}}_{P_t})) \leq \Theta(\mathcal{C}_{\mathcal{E}_o})(= \Theta(\mathcal{C}(\mathcal{E}_{P_t}))).$$

(**Part 2**) We prove $\mathcal{C}(\widetilde{\mathcal{E}}_2) \leq \Theta(\mathcal{C}(\mathcal{E}_0))$. In this part, we will generate each $R(\widetilde{P_i^r})$ by joining the matches of the cut edges $\{e_1, e_2, \ldots, e_s\}$ with $R(P_i^r)$. We suppose at least one cut edge is processed as otherwise it is trivial. Denote $\widetilde{P_i^r}[x]$ as $P_i^r \cup \{e_1, e_2, \ldots, e_x\}$. As each e_i is a cut edge for P_i^r, we have $R(\widetilde{P_i^r}[x]) \leq R(P_i^r)$ according to Equation 5.11.

Denote $\mathcal{C}(\widetilde{\mathcal{E}}_{\Delta\widetilde{P_i^r}})$ as the cost to generate $R(\widetilde{P_i^r})$ in the i[-th] round. We have:

$$C(\widetilde{\mathcal{E}}_2) = \sum_{i=1}^{t} \mathcal{C}(\widetilde{\mathcal{E}}_{\Delta\widetilde{P_i^r}}).$$

Let $\widetilde{P_i^r}[x]$ be $P_i^r \cup \{e_1, e_2, \ldots, e_x\}$, and specifically $\widetilde{P_i^r}[0] = P_i^r$. According to Equation 5.5, we have:

$$\mathcal{C}(\widetilde{\mathcal{E}}_{\Delta\widetilde{P_i^r}}) = \sum_{j=1}^{s} \mathcal{T}(e_j) + \sum_{x=0}^{s} \mathcal{T}(\widetilde{P_i^r}[x]).$$

Note that the number of cut edges is often small, and we treat s as a constant. In this sense, $\sum_{j=1}^{s} \mathcal{T}(e_j) = \Theta(M)$ and $\sum_{x=0}^{s} \mathcal{T}(\widetilde{P_i^r}[x]) = \Theta(\mathcal{T}(P_i^r))$, as $\mathcal{T}(\widetilde{P_i^r}[x]) \leq \Theta(\mathcal{T}(P_i^r))$ for each x given that $|R(\widetilde{P_i^r}[x])| \leq |R(P_i^r)|$ by Equation 5.11. Therefore:

$$C(\widetilde{\mathcal{E}}_2) = \Theta(M) + \sum_{i=0}^{t} \Theta(\mathcal{T}(P_{i_r})) \leq \Theta(\mathcal{C}(\mathcal{E}_o)).$$

According to Part 1 and Part 2, $\mathcal{C}(\widetilde{\mathcal{E}}) \leq \Theta(\mathcal{C}(\mathcal{E}_o))$, and apparently, $\mathcal{C}(\widetilde{\mathcal{E}}) \geq \mathcal{C}(\mathcal{E}_o)$. Therefore, $\mathcal{C}(\widetilde{\mathcal{E}}) = \Theta(\mathcal{C}(\mathcal{E}_o))$. □

We then show $\mathcal{C}(\mathcal{E}'_o) \leq \mathcal{C}(\widetilde{\mathcal{E}})$ under some practical relaxations.

Corollary 5.2. *Given a pattern graph P, and any P^l and P^r, such that $P = P^l \cup P^r$, we have $(P \setminus P^l) \subseteq P^r$.*

Proof. This is apparently true as $(P \setminus P^l)$ is the smallest P^r that satisfies $P = P^l \cup P^r$. □

Corollary 5.3. *Given a pattern graph P, and any left-join (resp. right-join) pattern P^l (resp. P^r), such that $P = P^l \cup P^r$, if there is no redundant node w.r.t. $P = P^l \cup P^r$, then we have $V(P^r) = V(P \setminus P^l)$.*

Proof. It is obvious that $V(P \setminus P^l) \subseteq V(P^r)$ by Corollary 5.2.

We then prove $V(P^r) \subseteq V(P \setminus P^l)$. For $\forall v \in V(P^r)$, we claim that $v \in V(P \setminus P^l)$. There are two cases. If $v \notin V(P^l)$, it is immediate that the claim is true. Otherwise, assume that $v \notin V(P \setminus P^l)$. Then $(P \setminus P^l) \subseteq P^r \setminus \{v\}$. Therefore, we must have $P = P^l \cup (P^r \setminus \{v\})$. In other words, v is a redundant node w.r.t. $P = P^l \cup P^r$. This draws a contradiction.

Based on the above cases, the corollary holds. □

Lemma 5.8. *If there is no redundant node w.r.t. $P_i = P_i^l \cup P_i^r$ for all $1 \leq i \leq |D_o| - 1$ in $\mathcal{E}_o = (D_o, J_o)$, then $\mathcal{C}(\mathcal{E}'_o) \leq \mathcal{C}(\widetilde{\mathcal{E}})$.*

Proof. Given $P_i = P_i^l \cup P_i^r$ in $\mathcal{E}_o$, and $\widetilde{P_i^r} = P_i^r \cup E_c(P_i^l, P_i^r)$, we further denote $\widetilde{P_i^{r*}} = (P_i \setminus P_i^l) \cup E_c(P_i^l, (P_i \setminus P_i^l))$.

We claim that $\widetilde{\mathcal{E}}$ must be within the searching space of the all-or-nothing strategy. It suffices to show that the join $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$ will be evaluated in the all-or-nothing strategy. Recall the process of the all-or-nothing strategy. When we have P_i^l, we will consider $\widetilde{P_i^{r*}}$ as the right-join pattern via the "all" strategy. In this sense, we simply prove the claim by showing that:

$$\widetilde{P_i^r} = \widetilde{P_i^{r*}}.$$

Since there is no redundant node w.r.t. $P_i = P_i^l \cup P_i^r$, according to Corollary 5.3. we have:

$$V(P_i^r) = V(P_i \setminus P_i^l).$$

Therefore, $V(\widetilde{P_i^{r*}}) = V(\widetilde{P_i^r})$.

We first show $\widetilde{P_i^r} \subseteq \widetilde{P_i^{r*}}$. $\forall e \in E(\widetilde{P_i^r})$, we have two cases: (1) if $e \in P_i^r$ and $e \notin E_c(P_i^l, P_i^r)$, then $e \notin P_i^l$. It is immediate that $e \in P_i \setminus P_i^l$. Hence, $e \in \widetilde{P_i^{r*}}$; (2) If $e \in E_c(P_i^l, P_i^r)$, let $e = (v_1, v_2), v_1, v_2 \in V(P')$. This means $v_1, v_2 \in V_c(P_i^l, P_i^r) = V(P_i^l) \cap V(P_i^r)$. On the other way, $V(P_i^l) \cap V(P_i^r) = V(P_i^l) \cap V(P' \setminus P_i^l) = V_c(P_i^l, P' \setminus P_i^l)$, which suggests $e = (v_1, v_2) \in E_c(P_i^l, P' \setminus P_i^l)$. Therefore, $e \in E(\widetilde{P_i^{r*}})$. With both cases, we have $\widetilde{P_i^r} \subseteq \widetilde{P_i^{r*}}$.

Then we show $\widetilde{P_i^{r*}} \subseteq \widetilde{P_i^r}$. $\forall e \in E(\widetilde{P_i^{r*}})$, it is obvious that $e \in P_i \setminus P_i^l$. By Corollary 5.2, we have $P_i \setminus P_i^l \subseteq P_i^r$, which suggests $e \in E(\widetilde{P_i^r})$. Thus, it holds that $\widetilde{P_i^{r*}} \subseteq \widetilde{P_i^r}$.

In conclusion, we have $\widetilde{P_i^r} = \widetilde{P_i^{r*}}$. This implies that $\widetilde{\mathcal{E}}$ must be within the searching space of the "all-or-nothing" strategy. While $\mathcal{E}_o'$ is the optimal solution in the space, it is immediate that $\mathcal{C}(\mathcal{E}_o') \leq \mathcal{C}(\widetilde{\mathcal{E}})$. □

Theorem 5.1. *If* $\mathcal{C}(\mathcal{E}_o) \geq \Theta(M)$ *and there is no redundant node w.r.t.* $P_i = P_i^l \cup P_i^r$ *for all* $1 \leq i \leq |D_o| - 1$ *in* $\mathcal{E}_o = (D_o, J_o)$*, then* $\mathcal{C}(\mathcal{E}_o') = \Theta(\mathcal{C}(\mathcal{E}_o))$

Proof. With Lemma 5.7 and Lemma 5.8, Theorem 5.1 holds. □

Discussion 5.3. *We show that the two conditions in Theorem 5.1 are practically reasonable. First,* $\mathcal{C}(\mathcal{E}_o) \geq \Theta(M)$*. Actually, the cost of the execution is often far larger than the size of the data graph. Second, no redundant node is involved. In practice, the involvements of redundant nodes usually result in more iterations, while the gain of such redundancies is rather limited.*

5.3 Performance Studies

In this subchapter, we show the experimental results for SEED. We rented a cluster from Amazon of up to 15 computing nodes including one master node and 14 slave

nodes and we used 10 slave nodes by default. The instance configurations of master and slave nodes are listed in Table 5.3. The hadoop configuration can be referred to that in Chapter 4.6.

Node Type	Instance	vCPU	Memory	Storage
master	m3.xlarge	4	15GB	2×40GB SSD
slave	c3.4xlarge	16	30GB	2×160GB SSD

Table 5.3: Amazon virtual instance configurations.

Datasets. We tested six real-world data graphs and two synthetic graphs (see Table 5.4). Among them, *lj*, *orkut* and *fs* were downloaded from SNAP (http://snap.stanford.edu), *yt* was downloaded from KONECT (http://konect.uni-koblenz.de), and *eu* and *uk* was downloaded from WEB (http://law.di.unimi.it). *pg21* and *pg29* are two power-law random graphs generated via [VL05] with $\beta = 2.1$ and $\beta = 2.9$, respectively. For each dataset, we list the number of nodes and edges (Note: m is for millions), and $T(G)$ - the time of constructing the SCP graph storage $\Phi^2(G)$ (Chapter 5.2.1). Note that the $T(G)$ for *pg21* and *pg29* are of no interest. The computation of SCP graph storage is query independent, and thus is considered as preprocessing step.

dataset	name	N	M	**T(G)**(s)
youtube	*yt*	3.22m	12.22m	27
eu-2015	*eu*	0.86m	16.14m	41
live-journal	*lj*	4.85m	42.85m	54
com-orkut	*orkut*	3.07m	117.19m	185
uk-2002	*uk*	18.52m	261.79m	841
friendster	*fs*	65.61m	1806.07m	2331
power-law($\beta = 2.1$)	*pg21*	10,000	50,000	-
power-law($\beta = 2.9$)	*pg29*	10,000	50,000	-

Table 5.4: Datasets used in the SEED Experiments.

Algorithms. We compared six algorithms:

- SEED: The SEED algorithm implemented in MapReduce with optimal bushy join plan and overlapping join units (Chapter 5.2.3).
- SEED-LD: SEED but with the (best) left-deep join plan.
- SEED-NO: SEED without overlapping join units.
- TT: The TwinTwigJoin algorithm implemented in MapReduce with all optimizations (Chapter 4).
- PSgL: The Pregel-based subgraph enumeration algorithm with all optimizations proposed by Shao et al. [SCC+14].

All algorithms were compiled with Java 1.7. We implemented SEED and all its variants and TT with Hadoop 2.6.0. All algorithms except PSgL are running on the Yarn framework. The authors of [SCC+14] kindly provided the codes for PSgL, implemented with Hadoop 1.2.0 on an old MapReduce framework. The performance gap between Yarn and old MapReduce is very small, hence the comparison between PSgL and the other algorithms is fair. We set the maximum running time to 4 hours. If a test did not stop within the time limit, or failed due to out-of-memory exceptions or other errors, we denoted the running time as INF. The time to compute the execution plan using Algorithm 5 is less than one second for all test cases, and thus has been omitted from the total processing time.

Queries. The seven queries denoted by q_1 to q_7 are illustrated in Figure 5.2 with the number of edges varying from 4 to 10 and the number of nodes varying from 4 to 6. We show the order of the nodes for automorphism resolution (Remark 2.1) under each query graph. Here, we have considered all queries (q_1 - q_4, q_7) except triangle in the literature. Note that triangle enumeration is used in this work as a preprocessing step to construct the SCP storage. We further added the queries q_5

and q_6 to demonstrate the advantages of our proposed techniques.

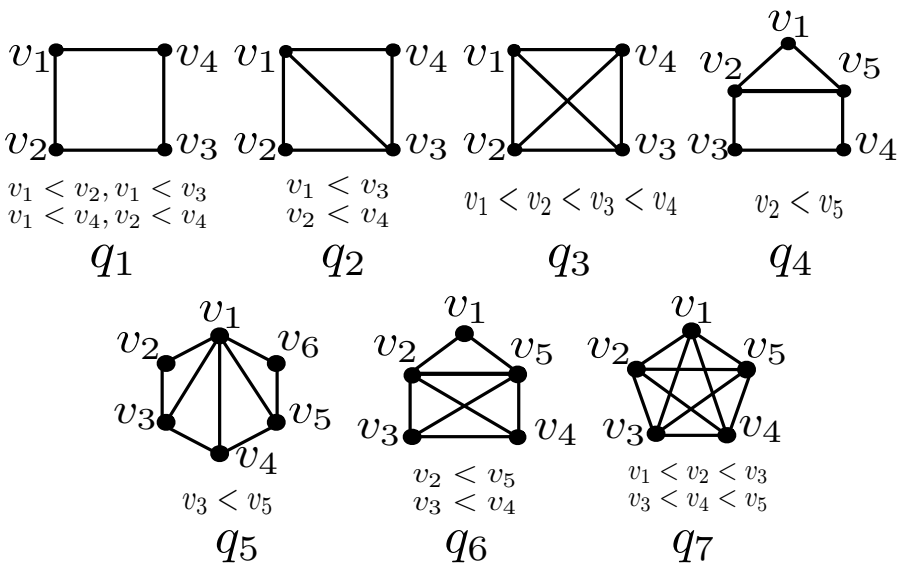
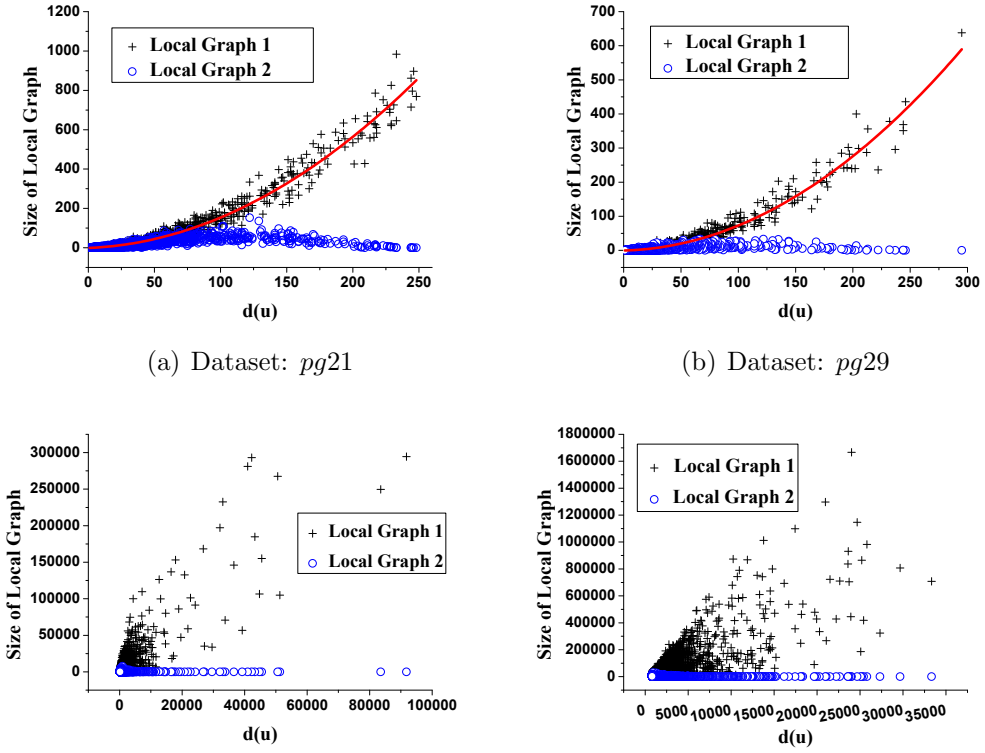


Figure 5.2: Queries used in the SEED experiment.

Exp-1: SCP Storage Mechanism. We measured $\Delta_u^{(1)}$ (Lemma 5.2) and $\Delta_u^{(2)}$ (Lemma 5.4) and their relationships with $d(u)$ for each data node u in both synthetic and real graphs. Results on the datasets *pg21*, *pg29*, *yt* and *orkut* are shown in Figure 5.3, where the X-axis depicts $d(u)$ and the Y-axis depicts $\Delta_u^{(1)}$ and $\Delta_u^{(2)}$. For clarity, we only reported the results for top-10000 largest nodes of *yt* and *orkut*. It is clear that $\Delta_u^{(1)}$ is much more skewed and larger than $\Delta_u^{(2)}$ in all tests. Particularly, we fit the curves for the $d(u)$-$\Delta_u^{(1)}$ relationship in the synthetic graphs. The results show that $\Delta_u^{(1)}$ is almost proportional to $d(u)^2$, which conforms with the theoretical result in Lemma 5.2. On the real graphs *yt* and *lj* (Figure 5.3(c), Figure 5.3(d)), besides skewness, we also observe that $\Delta_{max}^{(1)}$ on *lj* is nearly half the size of the data graph. As a result, $\Phi^1(G)$ can not scale to large data graphs. On the other side, the distributions of $\Delta_u^{(2)}$ are comparatively flat and small over $d(u)$ in all tests. The results validate that $\Phi^2(G)$ achieves good load balance and scalability, and hence is more storage-efficient. The local graph statistics for all other datasets are similar to those shown in Figure 5.3 and hence have been omitted.



(a) Dataset: *pg*21

(b) Dataset: *pg*29

(c) Dataset: *yt*

(d) Dataset: *orkut*

Figure 5.3: The results of Exp-1: SCP Storage Mechanism.

Exp-2: Bushy vs Left-deep. We compare the performance of SEED and SEED-LD using query q_5 on *yt* to test the advantage of using the bushy join plan. The plans $\mathcal{E}_1$ and $\mathcal{E}_2$ shown in Figure 3.1 illustrate the optimal execution plans for SEED-LD and SEED, respectively. Table 5.5 presents the experimental results, in which we observe a much better performance of SEED, compared to SEED-LD. We also show the output of mappers and reducers in each stage and compute the cost using Equation 5.5. The output of reduce^3 is not shown, as it is the final result and excluded in the cost. Observe that the algorithm with smaller cost always ends up with better performance. Clearly, SEED, with smaller cost, performs better than SEED-LD. The results are similar in the other datasets. We conclude that the optimal bushy join plan computed via Algorithm 5 outperforms the left-deep join

plan.

M/R	map^1	red^1	map^2	red^2	map^3	Cost	Time(s)
SEED	12.3	3.2	12.3	3.2	6.4	471.9	306
SEED-LD	12.3	3.2	15.5	6110.9	6123.9	31365.2	INF

Table 5.5: The results of Exp-2: Cost comparisons while enumerating q_5 on yt using SEED and SEED-LD (in millions).

As we mentioned in Remark 5.1, the plans $\mathcal{E}_1$ and $\mathcal{E}_2$ in Figure 3.1 are also the optimal execution plans computed via the ER model and the PR model, respectively. In Table 5.5, the outputs of reduce^2 of SEED and SEED-LD correspond to $|R(P_2^b)|$ and $|R(P_2^{ld})|$, and it is obvious that $|R(P_2^{ld})| \gg |R(P_2^b)|$. The results are consistent with our analysis in Remark 5.1 that the PR model offers more realistic cost estimation, which leads to better execution plan.

We chose q_5 in this experiment because of two reasons: (1) its optimal join plan is bushy; (2) the "optimal" join plans computed via ER model and PR model are different.

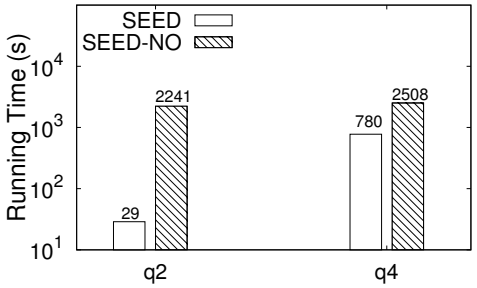
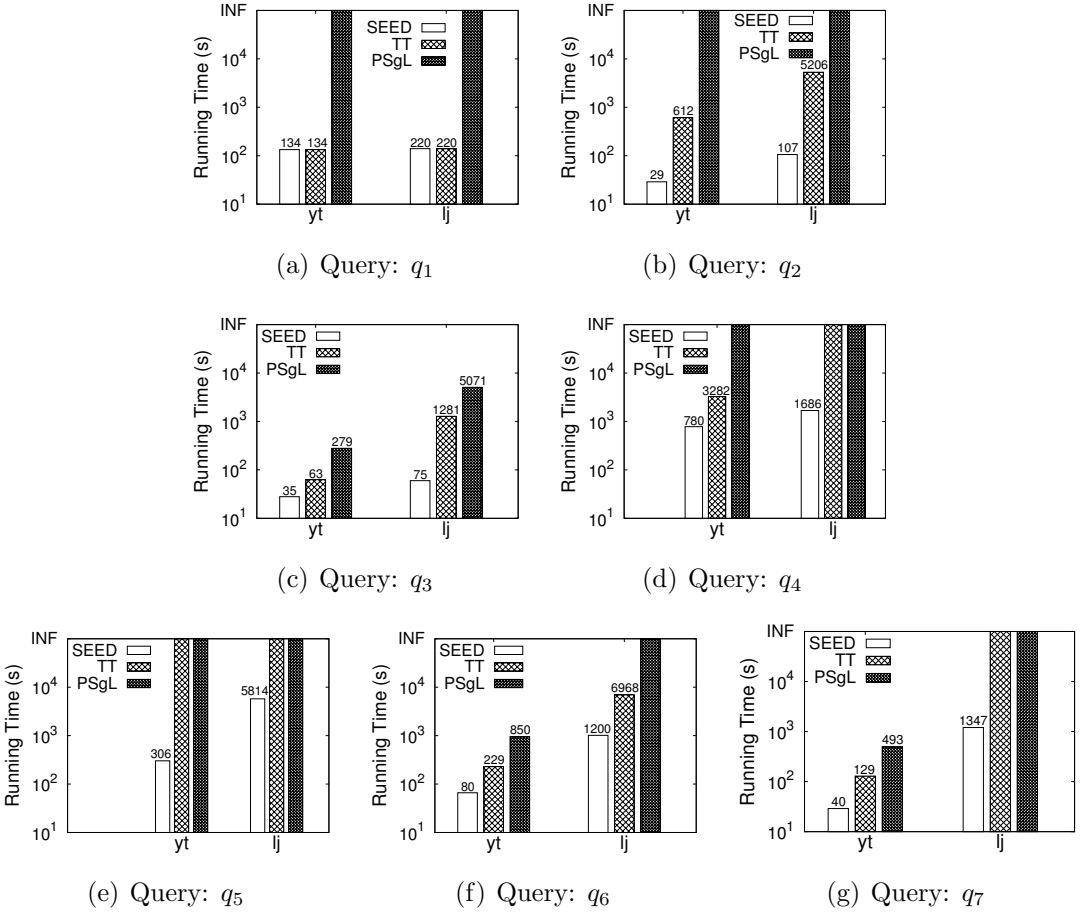


Figure 5.4: The results of Exp-3: SEED vs SEED-NO.

Exp-3: Overlapping Join Units. This experiment studied the benefit of overlapping the join units (Chapter 5.2.3). We processed the queries q_2 and q_4 on the dataset yt using SEED and SEED-NO. For q_2, SEED joins two triangles $p_0 = ((v_1, v_2), (v_2, v_3), (v_1, v_3))$ and $p_1 = ((v_1, v_3), (v_1, v_4), (v_3, v_4))$, while SEED-NO, without overlapping the join units, can only join p_0 to a TwinTwig

$p_1' = ((v_1, v_4), (v_3, v_4))$. Similarly, while processing q_4, SEED handles the triangle $((v_1, v_2), (v_1, v_5), (v_2, v_5))$ on the top, while SEED-NO can only use the TwinTwig $((v_1, v_2), (v_1, v_5))$. In practice, the triangle often renders much fewer results than the two-edge TwinTwig. Therefore, SEED outperforms SEED-NO, as shown in Figure 5.4. We obtained similar results on all queries other than q_1 (no overlapping exists), q_3, q_7 (clique itself is the join unit), and we only presented q_2 and q_4 as representatives.



(a) Query: q_1 (b) Query: q_2

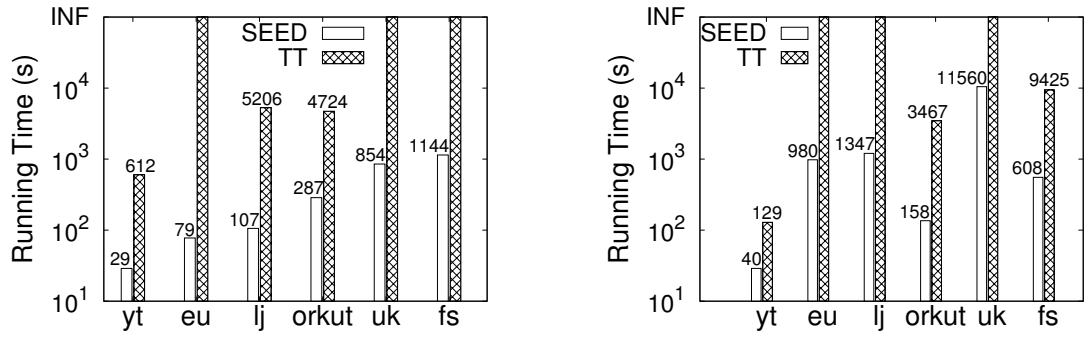
(c) Query: q_3 (d) Query: q_4

(e) Query: q_5 (f) Query: q_6 (g) Query: q_7

Figure 5.5: The results of Exp-4: Test against all queries.

Exp-4: Test against all queries. We compared SEED with TT and PSgL by enumerating all queries on *yt* and *lj*, and reported the results in Figure 5.5(a)-Figure 5.5(g). When enumerating q_1, SEED uses the same execution plan, and

hence has the same performance as TT, and they outperform PSgL. In all the other queries, SEED significantly outperforms TT, due to the use of clique as the join unit. For example, SEED is over 20× faster than TT while processing q_2 on both *yt* and *lj*, and over 15× faster than TT while processing q_3 on *lj*. Moreover, SEED processes complex queries such as q_4, q_5, q_6 and q_7 efficiently on both *yt* and *lj*. On the contrary, TT often runs out of time when querying on *lj*. PSgL can only process q_3 on *yt* and *lj*, and q_7 on *yt*, and in these cases, PSgL performs worse than TT. The reasons are two aspects. First, PSgL can be seen as StarJoin, which is already proven to be not better than TwinTwigJoin (Theorem 4.1). Second, the Pregel-based PSgL maintains all intermediate results in the main memory, and the numerous intermediate results produced in subgraph enumeration can exhaust the memory and cause unexpected termination of the algorithm. In conclusion, the proposed SEED algorithm significantly outperforms all existing algorithms, and TT also performs better than PSgL. Next we would exclude PSgL from the experiments, as it can only process simple queries on relatively small datasets.

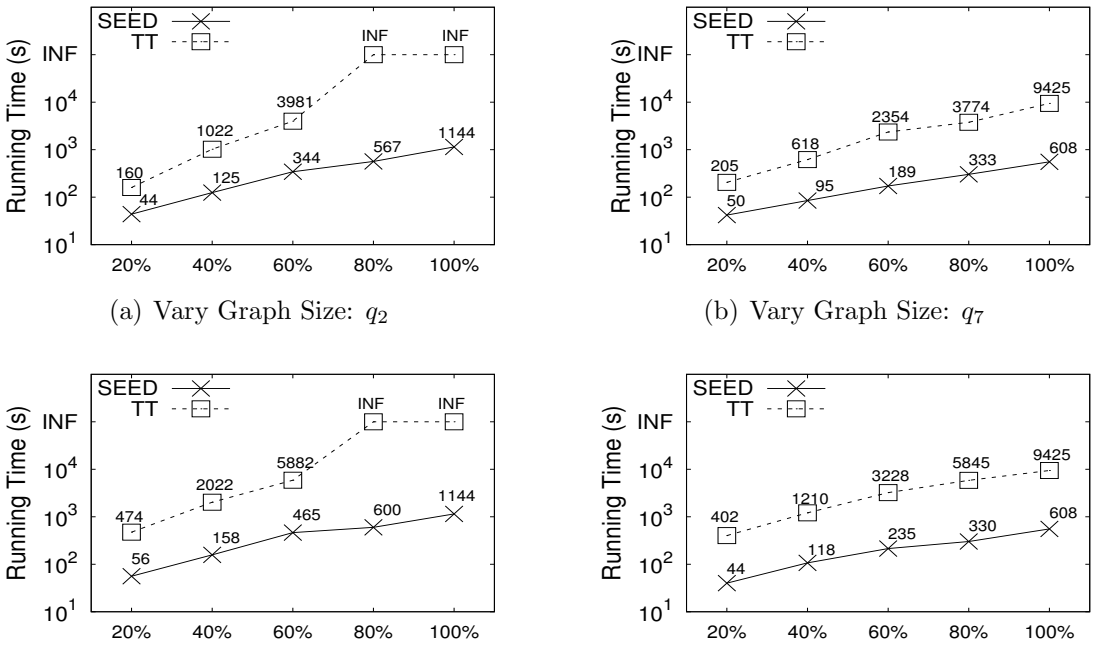


(a) Query: q_2 (b) Query: q_7

Figure 5.6: The results of Exp-5: Vary Datasets.

Exp-5: Vary Datasets. We compared SEED with TT by querying q_2 and q_7 on all datasets in order to show the advantages of SEED regarding different data properties. The results are shown in Figure 5.6(a)-Figure 5.6(b). In all tests, SEED

significantly outperforms TT, with the performance gain varying from an order of magnitude to over 50× (enumerating q_2 on *lj*). Specifically, SEED processes q_2 on the two largest datasets - *uk* and *lj*, in less than 20 minutes, while TT cannot terminate in the allowed time. This experiment demonstrates that SEED scales better for handling large data graphs due to the use of clique as the join unit, and the optimal bushy join plan with overlapping join units.



(a) Vary Graph Size: q_2

(b) Vary Graph Size: q_7

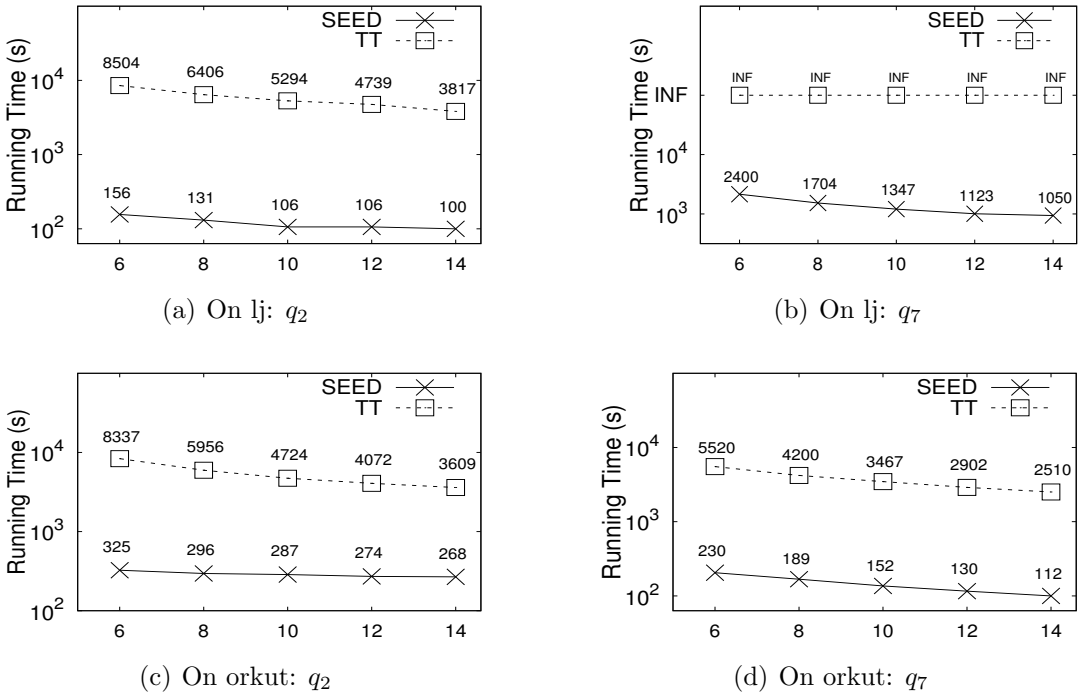
(c) Vary Avg. Degree: q_2

(d) Vary Avg. Degree: q_7

Figure 5.7: The results of Exp-6 and Exp-7: Vary graph properties.

Exp-6: 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_2 and q_7. The results are shown in Figure 5.7(a) and Figure 5.7(b) respectively. When the graph size increases, the running time of TT grows much more sharply than SEED. When the graph size is over 60%, only SEED finishes enumerating q_2 in the time limit. The test shows that SEED algorithm is more scalable than TwinTwigJoin.

Exp-7: Vary Average Degree. We fixed the set of nodes and randomly sampled 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_2 and q_7. The results are shown in Figure 5.7(c) and Figure 5.7(d) respectively. In Figure 5.7(d), SEED is 9, 10, 13, 17 and 15 times faster than TT when the average degree varies from 11 to 55, which shows the advantage of SEED for dense data graphs.



(a) On lj: q_2

(b) On lj: q_7

(c) On orkut: q_2

(d) On orkut: q_7

Figure 5.8: The results of Exp-8: Vary slave nodes.

Exp-8: 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 *orkut* datasets using queries q_2 and q_7. The test results are shown in Figure 5.8(a)-Figure 5.8(d) respectively. When the number of slave nodes increases, the running time of all algorithms decreases, and it drops more sharply when the number of slave nodes is small. On the one hand, increasing the number of slave nodes improves performance

by sharing the workload; on the other hand, it introduces extra communication costs from data transmissions among the slave nodes. As shown in Figure 5.8(b), even when 14 slave nodes are deployed, SEED is the only algorithm that can process q_7 on *lj*. We also tested other queries with various amount of slave nodes, and found curves similar to those in Figure 5.8(a)-Figure 5.8(d), thus the results have been omitted.

5.4 Chapter Conclusion

In this chapter, we proposed SEED, a scalable distributed subgraph enumeration algorithm. Compared to TwinTwigJoin (Chapter 4), SEED features with the following properties: (1) a novel SCP graph storage mechanism that allows using cliques, in addition to stars, as the join unit; (2) a comprehensive cost model based on the PR model; (3) a dynamic-programming algorithm to compute the optimal bushy join plan with overlapping join units. We have conducted extensive performance studies on real graphs with up to billions of edges, which shows that SEED outperforms the state-of-the-art works, including our TwinTwigJoin, by over an order of magnitude.

Chapter 6

Optimisation using Data Compression

We have discussed the optimal execution plan that aims at minimizing the cost while processing subgraph enumeration. Unfortunately, the task of subgraph enumeration for big graph often produces numerous intermediate results even while applying the optimal execution plan. Therefore, we introduce two data compression techniques, namely *compressed graph* (Chapter 6.1) and *clique compression* (Chapter 6.2) that further reduce the intermediate results by maintaining them in a compressed manner.

6.1 Compressed Graph

By aggregating data nodes that have the same neighbors into a *compressed node*, we construct a compressed graph, upon which the performance of subgraph enumeration can be further improved. In this subchapter, we 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 then discuss how to correctly process the query upon the compressed graph. For convenience, we only show details of adapting the TwinTwigJoin algorithm to the compressed graph, although similar technique can also be applied to SEED.

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

Definition 6.1. *(**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 6.2. *(**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 6.3. *(**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 6.4. *(**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 2.1).*

We then revise the total order $\prec$ (Definiton 2.1) as $\prec'$ in the data graph.

Definition 6.5. *(**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 6.1. *In the data graph G presented in Figure 6.1, 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 $\mathcal{S}_5 \prec^* \mathcal{S}_6 \prec^* \mathcal{S}_1 \prec^* \mathcal{S}_8 \prec^* \mathcal{S}_4$ by Definiton 6.4.*

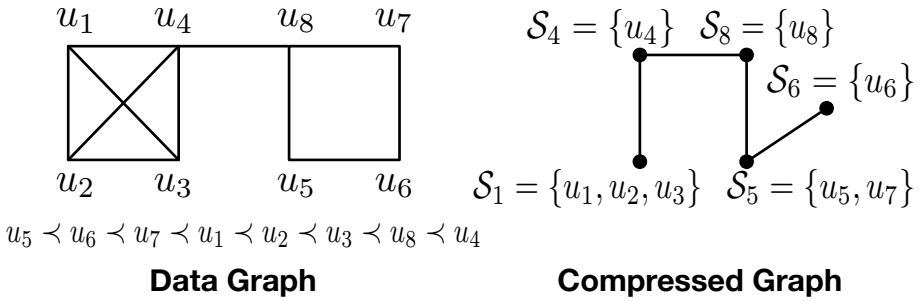


Figure 6.1: The Compressed node and compressed graph of the given data graph.

6.1.1 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* (ComprNodeGen) and *Compressed-Edge Binding* (ComprEdgeBind).

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. We describe the detailed algorithm in Algorithm 6, Algorithm 7 and Algorithm 8, where the symbols "$\boxtimes$", "$\therefore$" and "$\times$" are attached to indicate what kind of compressed node is being processed, as shown in Table 6.1.

The algorithm processes three rounds. In the first round (Algorithm 6), 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. If there are more than one node gathered in

Table 6.1: 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

Algorithm 6: ComprNodeGen-I(G stored as $\Phi^0(G)$ (Chapter 4.1))

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 $(u_{s_1};\ (\times, S = \{u_{s_1}\}))$;

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).

The second-round algorithm (Algorithm 7) 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

Algorithm 7: ComprNodeGen-II(G stored as $\Phi^0(G)$)

1 **function** map^2(**key**: u; **value**: $\mathcal{N}(u)$)

2 **if** $d(u) \leq \sqrt{M}$ **then** output $(\mathcal{N}(u);\ u)$;

3 **function** reduce^2(**key**: U; **value**: $S = \{u_{s_1}, u_{s_2}, \ldots, u_{s_k}\}$, where $u_{s_1} \prec u_{s_2} \cdots \prec u_{s_k}$)

4 **if** $|S| > 1$ **then**

5 output $(u_{s_1}; (\therefore, S))$;

6 **else** output $(u_{s_1};\ (\times, S = \{u_{s_1}\}))$;

associate the output with a "$\therefore$" (line 5), otherwise, a "$\times$" is attached similar to the first round (line 6). Note that in line 2 of Algorithm 6 and Algorithm 7, a degree threshold ($\sqrt{M}$) is applied, which will be explained later. For now we assume that there is no such a threshold in the algorithm. To summarize, the outputs of a node u in the first round and second round, denoted as $\mathsf{out}^1(u)$ and $\mathsf{out}^2(u)$, are related to the kinds of the compressed node of u. We have:

Proposition 6.1. *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***

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

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

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

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

Algorithm 8: ComprNodeGen-III(Outputs of Algorithm 6 and Algorithm 7)

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

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

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

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

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

(**If**) Let S_1 be the set of nodes aggregated on $\mathcal{N}[u]$ in reduce^1 (Algorithm 6) and S_2 be the set of nodes aggregated on $\mathcal{N}(u)$ in reduce^2 (Algorithm 7). 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 6) 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 6.1 and Definiton 6.2,

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 6) for u_{s_1}, the minimum node in S_1 (also the representative node of $\mathcal{S}(u)$). Therefore, we have $\mathsf{out}^1(u)$ as shown in case (ii).

(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 7), and we have out^2 as shown in case (ii).

(**Only If**) While $u = r_{\mathcal{S}(u)}$ and having $\mathsf{out}^1(u) = (u; (\boxtimes, \mathcal{S}(u)))$, it is apparent that $\mathcal{S}(u)$ is a clique compressed node. Otherwise, $\mathsf{out}^1(u) = \emptyset$. Clearly u does not belong to a trivial compressed node, as otherwise case iis 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. □

Based on Proposition 6.1, we generate the compressed nodes in the third round (Algorithm 8). The map^3 function reads $out^1(u)$ and $out^2(u)$ for each $u \in V(G)$, and outputs them in the form of $(u; (x, S))$, where $x \in \{\boxtimes, \therefore, \times\}$. 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 5-8) [1].

[1]Line 9-11 deal with the node with degree larger than the threshold.

Next we show the correctness of the ComprNodeGen algorithm.

Lemma 6.1. *Algorithm 8 returns each compressed node once and only once.*

Proof. It is clear that each trivial compressed node $\mathcal{S}(u) = \{u\}$ will be output in reduce^3 (Algorithm 8) 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 6.1, reduce^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. □

Example 6.2. *Following Example 6.1, we trace the outputs of node u_1 in each stage to show how* ComprNodeGen *runs. To start,* map^1 *outputs* $(\mathcal{N}[u_1] = \{u_1, u_2, u_3, u_4\}; u_1)$ *for* u_1. *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. In the second round,* map^2 *outputs* $(\mathcal{N}(u_1) = \{u_2, u_3, u_4\}; u_1)$. *On the key* $\{u_2, u_3, u_4\}$, reduce^2 *receives* $S = \{u_1\}$, *thus the algorithm outputs* $(u_1; (\times, \{u_1\}))$. *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. The procedures for the other nodes are similar, and we finally compute the five compressed nodes shown in Example 6.1.*

Handling large-degree node. The node u with $d(u) \geq \sqrt{M}$ will be directly assigned to a trivial compressed node in Algorithm 8 (line 9-11). We apply this threshold to avoid a very large key that can trigger overwhelmingly large sort cost in MapReduce. We show that it is highly impossible for such a node to have another equivalent node using the PR model. 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 9: 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.

function map_1^1(**key**: u; **value**: $\mathcal{N}(u)$)

output $(u;\ \mathcal{N}(u))$;

function map_2^1(**key**: $r_{\mathcal{S}}$; **value**: $\mathcal{S}$)

output $(r_{\mathcal{S}};\ \mathcal{S})$;

function reduce^1(**key**: u; **value**: $\{[\mathcal{S}(u)?], \mathcal{N}(u)\}$)

if $\mathcal{S}(u)$ exists in the value list **then** output $(\mathcal{S}(u);\ \mathcal{N}^o(\mathcal{S}(u)) = \mathcal{N}(u) \setminus \mathcal{S}(u))$;

function map_1^2(**key**: $\mathcal{S}$; **value**: $\mathcal{N}^o(\mathcal{S})$)

for each $u' \in \mathcal{N}^o(\mathcal{S})$ **do** output $(u';\ (\rightarrow, \mathcal{S}))$;

function map_2^2(**key**: $r_{\mathcal{S}}$; **value**: $\mathcal{S}$)

output $(r_{\mathcal{S}};\ (\in, \mathcal{S}))$;

function reduce^2(**key**: u; **value**: $\{[(\in, \mathcal{S}(u))?], [(\rightarrow, \mathcal{S}(u'))?]\}$)

if both $(\in, \mathcal{S}(u))$ and $(\rightarrow, \mathcal{S}(u'))$ exist in the value list **then**

  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 9. 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 9, 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 6.2. *Algorithm 9 returns all compressed edges.*

Proof. Given any compressed edge $(\mathcal{S}, \mathcal{S}') \in E(G^*)$, we show it is returned by Algorithm 9. Let $u = r_{\mathcal{S}}$. On the one hand, map_2^2 (Algorithm 9) 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 9) involves $(u; (\rightarrow, \mathcal{S}'))$ in the output. Finally, the above two key-value pairs arrive at reduce^2 (Algorithm 9), and the corresponding compressed edge is binded. □

Example 6.3. *We have generated the compressed nodes in Example 6.2, we use* $(\mathcal{S}_1, \mathcal{S}_4)$ *as an example to show how to bind the compressed edges via Algorithm 9. 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 6.1 and Lemma 6.2, 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 6.3. *Given the data graph* G*, the communication cost of constructing the compressed graph of* G *is* $O(M + N)$*, where* $M = |E(G)|$*, and* $N = |V(G)|$*.*

Proof. In MapReduce, communication cost is triggered by transferring the output data of each mapper to the reducer. In Algorithm 6 and Algorithm 7, map^1 and map^2 output the neighbors for each node, and the cost is $O(M)$, and map^3 in Algorithm 8 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 9, 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. □

6.1.2 Querying the compressed graph

For convenience, we follow TwinTwigJoin to discuss how to process subgraph enumeration on the compressed graph. We first introduce the *Compressed Match*.

Definition 6.6. *(**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 $f^*(v_i).clique = true$ 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 6.5). Clearly, a compressed match can represent multiple matches.

Example 6.4. *Consider the square pattern graph given in Figure 2.1. There are four matches of the square in the data graph presented in Figure 6.1. 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^*.

Recall that in TwinTwigJoin, with the pattern graph decomposing into a set of TwinTwigs $\mathcal{D}(P) = \{p_0, p_1, \ldots, p_t\}$, 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 of TwinTwigJoin 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 $(u; G_u^0)$ 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 4.6.

Denote $R^*(P)$ as the set of compressed matches. Given the same pattern decomposition, the SubgEnumCompr iteratively processes the 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 TwinTwigJoin, 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 the TwinTwig p_i. Finally, in step (4), SubgEnumCompr filters the results based on the *Size-Limitation* and *Order-Preservation* constraints in Definiton 6.6.

We first discuss how SubgEnumCompr computes $R^*(p)$ in step (3) for a TwinTwig p on the compressed graph. Note that the idea can be applied handle star and clique, so that we can also adapt SEED to 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 $p = ((v_0, v_1), (v_0, v_2))$, and suppose G^* is stored in the form of $(\mathcal{S}; \mathcal{N}^*(\mathcal{S}))$ for each $\mathcal{S} \in V(G^*)$. We focus on computing the compressed matches of p for each $\mathcal{S}$ independently. Denote $R^*_{\mathcal{S}}(p) = \{f^* \mid f^*(v_0) = \mathcal{S}\}$ as the $\mathcal{S}$-fixed compressed matches. It is clear that $R^*(p) = \bigcup_{\mathcal{S} \in V(G^*)} R^*_{\mathcal{S}}(p)$. We show that we can correctly compute $R^*_{\mathcal{S}}(p)$ on each $\mathcal{S}$ (and hence $R^*(p)$) using Algorithm 10.

Corollary 6.1. *Given a* TwinTwig p *and any compressed node* $\mathcal{S} \in V(G^*)$*, Algorithm 10 correctly computes* $R^*_{\mathcal{S}}(p)$*.*

Proof. Consider match of p - (u_0, u_1, u_2), and the corresponding compressed match

Algorithm 10: 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 $\mathcal{S}$-fixed compressed matches of p.

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

$(\mathcal{S}(u_0), \mathcal{S}(u_1), \mathcal{S}(u_2))$ that has $\mathcal{S}(u_0) = \mathcal{S}$. 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, $u_0, u_1, u_2 \in \mathcal{S}$, and we must have $\mathcal{S}$.clique=true due to $(u_0, u_1) \in E(G)$. This compressed match is handled in line 3 in Algorithm 10.

- $\mathcal{S} = \mathcal{S}(u_1)$ 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 6.

- $\mathcal{S}(u_1) = \mathcal{S}(u_2)$ and $\mathcal{S}(u_1) \neq \mathcal{S}$. Note that $\mathcal{S}(u_1) \in \mathcal{N}^*(\mathcal{S})$, and this case is covered in line 7.

- $\mathcal{S} \neq \mathcal{S}(u_1) \neq \mathcal{S}(u_2)$. Both compressed nodes are $\mathcal{S}$'s neighbors. Algorithm 10 covers this case in line 9 by enumerating the pairs of compressed nodes in $\mathcal{N}*(\mathcal{S})$.

Summarizing the above cases, Algorithm 10 returns all $R^*_{\mathcal{S}}(p)$, which completes the proof. □

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

Lemma 6.4. *Given the pattern graph P, and the compressed graph G^*,* SubgEnumCompr *correctly computes all compressed matches.*

Proof. 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 6.1.

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 6.1, 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. □

Example 6.5. *Consider the square pattern in Figure 2.1. 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 Figure 6.1. 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 10, 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 6.2.

Table 6.2: 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)

6.2 Clique Compression

To start this section, let us consider a motivating example.

Example 6.6. *We find a large clique with 943 nodes in the uk dataset used in our experiment in Table 6.3, which alone contributes to* $\binom{943}{5} \approx 6 \times 10^{12}$ *matches for a* 5*-clique, and causes huge burden on storage and communication. Alternatively, we can encode all these results using the nodes of the large clique itself, and this costs linear space to the number of nodes in the clique.*

This example motivates us to consider clique compression, aiming at reducing the cost of transferring and maintaining the intermediate results. In order to do so, we compute a set of non-overlapping cliques in the data graph G as a preprocessing step. Let p^k be a clique of k nodes. In query processing, when p^k is considered as a join unit, instead of computing all the matches of p^k directly, we represent the matches in a compressed way, and we also try to maintain the compressed matches in future joins. Note that we only apply clique compression to SEED where clique can be the join unit. In the following, we first show how to precompute the non-

overlapping cliques, followed by discussing the way of compressing the matches of p^k. Finally, we introduce how to process joins with the compressed results.

Algorithm 11: Clique-Search(data graph G)

Input : G, the data graph.

Output : A set of non-overlapping cliques.

```
1 G' ← G;  S ← ∅;
2 while G' is not empty do
3     u ← The node with largest degree in G';
4     K ← A maximal clique containing u in G';
5     if |V(K)| > thresh then
6         S = S ∪ {K};
7     G' ← G' \ K;
8 return S;
```

6.2.1 Clique Precomputation

As a preprocessing step, we compute a set of non-overlapping (by nodes) cliques $S = \{K_1, K_2, \ldots, K_s\}$ in the data graph G. We show the greedy algorithm to compute S in Algorithm 11. Each time we select a node u with the largest degree from G, compute a maximal clique containing u in G, add the clique into S if its size is larger than a threshold (e.g., 50) and remove it from G. We repeat the process until all nodes are removed from G. After computing S, we index all the cliques on each machine in the cluster (e.g. using "Distributed Cache" in MapReduce). Specifically, we maintain a map $\mathcal{M}$ in each machine, so that we can use $\mathcal{M}(u)$ to determine the clique that a node u ($u \in V(G)$) belongs to in constant time. Let $\mathcal{M}(u) = \emptyset$ if u does not belong to any clique in S. The space used to index the

cliques is small since we only need to index the nodes in each clique. We show in the experiment that the overhead of clique precomputation is relatively small, and it contributes to improving the performance of SEED, especially when the data graph contains some large cliques.

6.2.2 Online Clique Compression

During query processing, suppose a k-clique p^k is involved in the join, where $V(p^k) = \{v_1^\kappa, v_2^\kappa, \ldots, v_k^\kappa\}$ and $v_1^\kappa < v_2^\kappa < \cdots < v_k^\kappa$ (for symmetry breaking (Remark 2.1)). We compress the matches of p^k as follows. In each local graph $G_u \in \Phi^2(G)$ (Chapter 5.2.1), we divide the nodes in $V(G_u^2)$ into two parts, namely, the clique nodes V_u^c and the non-clique nodes V_u^n. Here $V_u^c = \{u' | u' \in V(G_u^2) \setminus \{u\}, \mathcal{M}(u') = \mathcal{M}(u)\}$ is the set of nodes in G_u^2 that belong to the same clique as u in S, and $V_u^n = V(G_u) \setminus V_u^c$. Note that we have $u \in V_u^n$ for the ease of presentation. Specifically, when $\mathcal{M}(u) = \emptyset$, we have $V_u^c = \emptyset$ and $V_u^n = V(G_u)$. The nodes in both set are rearranged via the data node orders (Definiton 2.1). With the two different types of nodes, a compressed match, which represents multiple matches of the k-clique, is denoted as $f^* = (f^c, f^n)$, where $f^c = (f^c.V, f^c.U) = (\{v_1^c, v_2^c, \ldots, v_s^c\}, \{u_1^c, u_2^c, \ldots, u_t^c\})$ is the compressed part of the match and $f^n = (f^n.V, f^n.U) = (\{v_1^n, v_2^n, \ldots, v_{k-s}^n\}, \{u_1^n, u_2^n, \ldots, u_{k-s}^n\})$ is the non-compressed part. We also regard f^n as a *partial match*, where $f^n(v_n^i) = u_i^n$. Here, the following five constraints must be satisfied:

- C_1: $u \in f^n.U$.
- C_2: $f^n.U \subseteq V_u^n$ and the nodes in $f^n.U$ must form a clique in G_u^2.
- C_3: $f^c.U \subseteq V_u^c$ and every node in $f^c.U$ is adjacent to all nodes in $f^n.U$ in G_u^2.
- C_4: $|f^c.V| \leq |f^c.U|$.

- C_5: $f^c.V \cup f^n.V = V(p^k)$.

In this way, a compressed match (f^c, f^n) represents $\binom{t}{s}$ matches of a k-clique, that is, the $k-s$ nodes $f^n.U = \{u_1^n, u_2^n, \ldots, u_{k-s}^n\}$ and every combination of s nodes in $f^c.U = \{u_1^c, u_2^c, \ldots, u_t^c\}$ recover a match. Note that C_1 restricts that u must be in the match, which is applied to avoid duplicates. For example, consider a 5-clique $\{u_1, u_2, u_3, u_4, u_5\}$ as the data graph, and a 4-clique p^4 as the pattern graph. Without C_1, the match (u_2, u_3, u_4, u_5) will be computed twice in both G_{u_1} and G_{u_2}. However, this match will be removed from G_{u_1} by C_1 as u_1 does not appear in the match. Considering the Order-Preservation constraint (Remark 2.1), we actually match the smallest node in p^k to u (note that u is the smallest node in G_u). Thus, C_1 can be accordingly written as $f^n(v_1^\kappa) = u$.

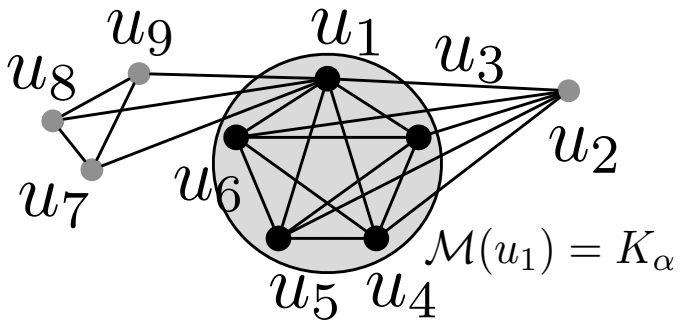


Figure 6.2: The local graph of u_1, and clique compression.

Example 6.7. *In Figure 6.2, we show the local graph G_{u_1}. Note that all nodes except u_1 have neighbors not presented in G_{u_1} and these nodes are already arranged by their orders (Definiton 2.1) in the data graph. The shadowed circle highlights a 5-clique K_α that u_1 belongs to. Observe that u_2 forms a larger clique with K_α but it does not belong to it. This can happen when we assign u_2 to the other larger clique. Thus, we have the clique nodes $V_{u_1}^c = \{u_3, u_4, u_5, u_6\}$, and the non-clique nodes $V_{u_1}^n = \{u_1, u_2, u_7, u_8, u_9\}$. Hereunder, we show the compressed matches of the 4-clique p^4 in G_{u_1}:*

#	$f^c.V$	$f^c.U$	$f^n.V$	$f^n.U$	# matches
f_1^*	$\{v_2^\kappa, v_3^\kappa, v_4^\kappa\}$	$\{u_3, u_4, u_5, u_6\}$	$\{v_1^\kappa\}$	$\{u_1\}$	4
f_2^*	$\{v_3^\kappa, v_4^\kappa\}$	$\{u_3, u_4, u_5, u_6\}$	$\{v_1^\kappa, v_2^\kappa\}$	$\{u_1, u_2\}$	6
f_3^*	$\emptyset$	$\emptyset$	$\{v_1^\kappa, v_2^\kappa, v_3^\kappa, v_4^\kappa\}$	$\{u_1, u_7, u_8, u_9\}$	1

Considering the Order-Preservation constraint (Remark 2.1), the data node sequence that matches p^4 must be arranged in the increasing order. In f_1^*, u_1 together with each 3-combination of $f^c.U$ (increasing order) recover a match of the 4-clique, and thus f_1^* compresses $\binom{4}{3} = 4$ results. Similarly, f_2^* compresses $\binom{4}{2} = 6$ results. Additionally, $f^* f_3^*$ corresponds to a non-compressed match. As a whole, we use 3 compressed matches to represent 11 results.

The algorithm to compute all compressed k-cliques in a certain G_u^2 is shown in Algorithm 12. It is worth noting that Algorithm 12 can handle the non-compressed (unoptimized) case by letting $\mathcal{M}(u) = \emptyset$ for each $u \in V(G)$. Before moving forward to the algorithm, we define the *clique neighbors* and *common clique neighbors* as follows:

Given $u' \in V_u^n$, the *clique neighbors* of u', denoted as $CN(u')$, are the nodes in V_u^c that are adjacent to u' in G_u, that is, $CN(u') = \{u'' | u'' \in V_u^c \wedge (u', u'') \in E(G_u)\}$, and the *common clique neighbors* of a set of data nodes U', denoted as $CCN(U') = \bigcap_{u' \in U'} CN(u')$.

In Algorithm 12, we first assign the clique nodes V_u^c and the non-clique nodes V_u^n in line 2-3, and we use $V_u^n[i]$ to denote the i[-th] (start from 1) node in V_u^n. In line 4, we report $(f^c = (V(p^k) \setminus \{v_1^\kappa\}, V_u^c), f^n = (\{v_1^\kappa\}, \{u\}))$ as a fully compressed results. Clearly, u and each $k-1$ combination of V_u^c recover a match of p^k. We then call the recursive function CompressedCliqueRec to further generate the compressed matches (line 5). In the recursive function, we use U' to record the non-clique nodes that form a clique in G_u, which will be expanded in each recursive call, and be assigned to $f^n.U$ afterwards.

Algorithm 12: CompressedClique(p^k, G_u)

Input : p^k, the k-clique, where $V(p^k) = \{v_1^\kappa, v_2^\kappa, \ldots, v_k^\kappa\}$,
G_u, the local graph of a certain $u \in V(G)$.

Output: A set of compressed matches of p^k.

$\mathcal{F} \leftarrow \emptyset$;
$V_u^c \leftarrow \{u' | u' \in V(G_u) \setminus \{u\} \wedge \mathcal{M}(u') = \mathcal{M}(u)\}$;
$V_u^n \leftarrow V(G_u) \setminus V_u^c$;
if $|V_u| \geq k-1$ **then** $\mathcal{F} \leftarrow \mathcal{F} \cup (f^c = (V(p^k) \setminus \{v_1^\kappa\}, V_u^c), f^n = (\{v_1^\kappa\}, \{u\}))$;
CompressedCliqueRec($p^k,\ V_u^c,\ V_u^n \setminus \{u\},\ \mathcal{F},\ 1,\ \{u\}$)
return $\mathcal{F}$;
function CompressedCliqueRec($p^k,\ V_u^c,\ V_u^n,\ \mathcal{F},\ i,\ U'$)
foreach $j \in [i, |V_u^n|]$ **do**
  $u' \leftarrow V_u^n[j]$;
  if u' forms a clique with U' in G_u **then**
    $U' \leftarrow U' \cup \{u'\}$;
    Initialize the compressed match ($f^c = \{\emptyset, \emptyset\}, f^n = \{\emptyset, \emptyset\}$);
    if ($|U'| < k$) **then**
      $f^c.U \leftarrow CCN(U')$;
    if $|U'| + |f^c.U| \geq k$ **then**
      foreach $\{v_1^n = v_1^\kappa, v_2^n, \ldots, v_{k-s}^n\} \subseteq V(p^k)$ s.t. $v_1^n < v_2^n < \cdots v_{k-s}^n$, where $|U'| = k - s$ **do**
        $f^n.V \leftarrow \{v_1^n, v_2^n, \ldots, v_{k-s}^n\}$;
        $f^c.V \leftarrow V(p^k) \setminus f^n.V$;
        $f^n.U \leftarrow U'$;
        $\mathcal{F} \leftarrow \mathcal{F} \cup (f^c = (f^c.V, f^c.U), f^n = (f^n.V, f^n.U))$;
      if $|U'| < k \wedge j \neq |V_u^n|$ **then**
        CompressedCliqueRec($f^c.U,\ V_u^n \setminus U',\ \mathcal{F}, j+1,\ U'$);

The algorithm then proceeds by checking the above five constraints for a compressed match. C_1 is guaranteed to be satisfied while we fix u in $f^n.U$ in line 4. For each non-clique nodes that have not been visited (line 8), we verify that if it forms a larger clique with U' (line 10) in order to guarantee that C_2 is satisfied. U' is expanded by involving the qualified node (line 11). If $|U'| < k$, we let $f^c.U = CCN(U')$ (line 14) to satisfy C_3. Otherwise, U' must have included all nodes for a full-matched subgraph, and we simply leave $f^c.U = \emptyset$. The condition $|U'| + |f^c.U| \geq k$ in line 15 guarantees C_4 and C_5 are satisfied, and once does, we follow the procedure in line 16-20 to generate the compressed matches. For each $k - s$ nodes from $V(p^k)$ that involves v_1 (line 16), we assign them to $f^n.V$ (line 17), and then we obtain a partial match $f^n = (f^n.V, f^n.U = U')$. Thus, we construct a compressed match $(f^c = (f^c.V, f^c.U), f^n = (f^n.V, f^n.U))$ (line 20). We recursively call CompressedCliqueRec as $|U'| < k$ (line 22).

Example 6.8. *In Example 6.7, we have shown the compressed matches f_1^*, f_2^* and f_3^* of p^4 in G_{u_1}. According to Algorithm 12, f_1^* is computed in line 4. Now that we have $V_u^n = \{u_2, u_7, u_8, u_9\}$ (u_1 is excluded). In the first recursive call, we have $U' = \{u_1, u_2\}$ after adding u_2 (line 11). It is clear that $CCN(U') = \{u_3, u_4, u_5, u_6\}$. As u_1 has already been fixed to match v_1^κ, we iterate the nodes in $\{v_2^\kappa, v_3^\kappa, v_4^\kappa\}$ to match u_2. When v_2^κ is matched to u_2, we obtain f_2^*(line 16). After processing u_2, we move to the next node in V_u^n , that is u_7, and the recursive function will return f_3^* as a non-compressed match. (line 13).*

6.2.3 Online Join Processing

We apply clique compression to SEED, which follows Algorithm 4 to process joins, but replace each match f in Algorithm 4 as a compressed match $f^* = (f^c, f^n)$. We correspondingly revise the map and reduce functions to handle the compressed

match. Note that here we generalize the concept of "compressed match", which not only represents a compressed match of a k-clique, but also the compressed join results produced in each round (Details are in Algorithm 14). For a non-compressed match, we simply let $f^c.V = f^c.U = \emptyset$. The main challenge is that, when a compressed match (f^c, f^n) is involved in a join, we do not need to immediately recover all matches from (f^c, f^n). Instead we try to maintain its compressed part f^c as much as possible. We call this process partial expansion. Given a compressed match (f^c, f^n), suppose it is involved in a join with join attributes V_{join}, the process of partial expansion is shown in lines 12-20 in Algorithm 13. We first compute the non-clique join attributes V^n_{join} and its corresponding match U^n_{join} (lines 13-14). Then we compute V^c_{join} - the set of join attributes that need to be expanded in the clique part f^c (line 15). Line 16 enumerates all matches U^c_{join} of V^c_{join} in f^c. For each U^c_{join}, we output a key-value pair (line 20) where the key is computed as $U^c_{join} \cup U^n_{join}$ (line 17) and the value is a compressed match (f^c_{out}, f^n_{out}) by moving the original match of V^c_{join} from f^c to f^n (line 18-19). The revised map^i procedure is shown in Algorithm 13 to replace map^i in Algorithm 4.

Example 6.9. *Suppose $f^*_1 = (f^n_1, f^c_1)$ in Example 6.7 is involved in the join with $V_{join} = \{v^\kappa_1, v^\kappa_3\}$. We have $f^n_1.V = \{v^\kappa_1\}$, $f^n_1.U = \{u_1\}$, $f^c_1.V = \{v^\kappa_2, v^\kappa_3, v^\kappa_4\}$ and $f^c_1.U = \{u_3, u_4, u_5, u_6\}$. We first compute $V^n_{join} = f^n_1.V \cup V_{join} = \{v^\kappa_1\}$ and $U^n_{join} = \{u_1\}$. Then we have $V^c_{join} = f^c.V \cap V_{join} = \{v^\kappa_3\}$. Consequently, we should partially expand f^c_1 by taking a node out of $f^c.U$ as U^c_{join}. We first take u_3, and the join key is $\{u_1, u_3\}$, then we compute $f^c_{out} = (\{v^\kappa_2, v^\kappa_4\}, \{u_4, u_5, u_6\})$ and $f^n_{out} = (\{v^\kappa_1, v^\kappa_3\}, \{u_1, u_3\})$. Ultimately, the key-value pair $(\{u_1, u_3\}; (f^c_{out}, f^n_{out}))$ is generated.*

Algorithm 14 presents the detailed algorithm of the revised reduce^i, which takes the compressed matches as inputs, and output the join results in the same com-

Algorithm 13: map^i(**key**: $\emptyset$; **value**: either compressed matches $(f^c, f^n) \in R(P'_j)$ and $(h^c, h^n) \in R(P'_s)$ for some $j < i, s < i$ or $G_u \in \Phi(G)$)

$V_{join} \leftarrow V(P'_j) \cap V(P'_s)$;
if P'_j is a star **then** genJoinUnit(P'_j, G_u, V_{join});
else if P'_j is a clique **then** genCompressedClique(P'_j, G_u, V_{join});
else PartialExpansion(f^c, f^n, V_{join});
if P'_s is a star **then** genJoinUnit(P'_s, G_u, V_{join});
else if P'_s is a clique **then** genCompressedClique(P'_s, G_u, V_{join});
else PartialExpansion(h^c, h^n, V_{join});
function genCompressedClique(p^k, G_u, V_{join})
$\mathcal{F} \leftarrow$ CompressedClique(p^k, G_u);
foreach $(f^c, f^n) \in \mathcal{F}$ **do**
    PartialExpansion(f^c, f^n, V_{join});
function PartialExpansion(f^c, f^n, V_{join})
$V^n_{join} = \{v^n_1, v^n_2, \ldots, v^n_p\} \leftarrow f^n.V \cap V_{join}$;
$U^n_{join} \leftarrow \{f^n(v^n_1), f^n(v^n_2), \ldots, f^n(v^n_p)\}$;
$V^c_{join} \leftarrow f^c.V \cap V_{join}$;
foreach $U^c_{join} \subseteq f^c.U$ s.t. $|U^c_{join}| = |V^c_{join}|$ **do**
    $key \leftarrow U^c_{join} \cup U^n_{join}$;
    $f^c_{out} \leftarrow (f^c.V \setminus V^c_{join}, f^c.U \setminus U^c_{join})$;
    $f^n_{out} \leftarrow (f^n.V \cup V^c_{join}, f^n.U \cup U^c_{join})$;
    output $(key; (f^c_{out}, f^n_{out}))$;

pressed way, which can hence be treated as compressed matches in a future join. Given two compressed results $h_1 = (h^c_1, h^n_1)$ and $h_2 = (h^c_2, h^n_2)$, we say h_1 has larger compression power than h_2 if $|h^c_1.V| > |h^c_2.V|$. For the two compressed results from the current join patterns, the idea is to expand the one with smaller compression

Algorithm 14: reduce^i(**key**: U_{join}; **value**: Two sets of compressed matches H_1 and H_2)

```
// We assume |h_1^c.V| ≥ |h_2^c.V| (w.l.g.).
1  foreach h_1 = (h_1^c, h_1^n) ∈ H_1, h_2 = (h_2^c, h_2^n) ∈ H_2 s.t.
   (h_1^n.U \ U_join) ∪ (h_2^n.U \ U_join) = ∅ do
2      if h_2^c.V = ∅ then
3          f^n.V = h_1^n.V ∪ h_2^n.V; f^n.U = h_1^n.U ∪ h_2^n.U;
4          f^c.V = h_1^c.V; f^c.U = h_1^c.U;
5          output (∅; (f^c, f^n));
6      else
           // We need to expand h_2 that has smaller compression.
7          f^c.V = h_1^c.V; f^c.U = h_1^c.U;
8          f^n.V = h_1^n.V ∪ h_2^n.V ∪ h_2^c.V;
9          foreach U' ⊆ h_2^c.U s.t. |U'| = |h_2^c.V| do
10             f^n.U = h_1^n.U ∪ h_2^n.U ∪ U' ;
11             output (∅; (f^c, f^n));
```

while keeping the other. In reduce^i, we process the join of h_1 and h_2 for each $h_1 \in H_1$ and $h_2 \in H_2$ (line 1). Without loss of generality, we assume that h_1 has larger compression power. If h_2 is not compressed (line 2), we simply union h_1^n and h_2^n to form the non-compressed part f^n (line 3), keep h_1^c in the compressed part f^c (line 4), and output the results. Otherwise, we have to expand h_2. We still keep h_1^c in f^c (line 8), while f^n now includes not only h_1^n and h_2^n, but a part from the expansion of h_2^c (line 8). We hence iterate every $U' \subseteq h_2^c.U$ with $|U'| = |h_2^c.V|$, compute $f^n.U = h_1^n.U \cup h_2^n.U \cup U'$ (line 10), and output the compressed results (line 11). In this way, we compress the results in each join, and they can be treated

just like the compressed matches of the k-clique in the future join (Algorithm 13).

6.3 Performance Studies

We present the experiment results that verifies the effectiveness of the data compression techniques. The MapReduce setting is the same to that in the SEED experiment (Chapter 5.3).

Datasets. We present the dataset in Table 6.3. The number of edges and nodes for each data can be found in Table 4.1 and Table 5.4. The "r_v" and "r_e" columns in the table represent the node-compression ratio of and edge-compression ratio for compressed graph, 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 [RW15]. Clearly, our MapReduce implementation is far more efficient than Ren's algorithm. The "time / s (C)" column denotes the time to precompute and index the clique for each dataset.

Table 6.3: Datasets used in the data-compression experiments.

dataset	**name**	r_v (%)	r_e (%)	time/s (MR)	time/s (Ren)	time/s (C)
youtube	*yt*	45.48	79.47	165	467	58
eu-2015	*eu*	56.87	42.14	170	1056	129
live-journal	*lj*	77.37	95.16	179	1332	170
com-orkut	*orkut*	97.73	99.94	222	7995	345
indochina-2004	*indo*	**50.28**	**39.44**	124	INF	897
uk-2002	*uk*	**50.82**	**39.96**	264	INF	1270
friendster	*fs*	65.31	99.66	2138	INF	368

Algorithms. We implemented and compared the following algorithms:

- TT: The TwinTwigJoin algorithm with all optimizations.

- TT+C: TT on the compressed graph (Chapter 6.1).
- SEED: The most optimized SEED algorithm.
- SEED+C: SEED with clique compression (Chapter 6.2).

All algorithms were implemented using Hadoop (version 2.6.2) with Java 1.7. We set the maximum running time to 4 hours. If a test did not stop within the time limit, or failed due to out-of-memory exceptions or other errors, we denoted the running time as INF.

Queries. We use four most representative queries q_1 to q_4 from our previous experiments as shown in Figure 6.3 to test the data compression technique.

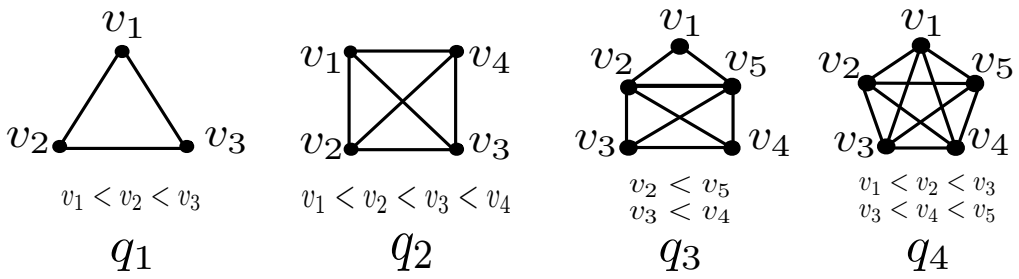


Figure 6.3: Queries for data compression.

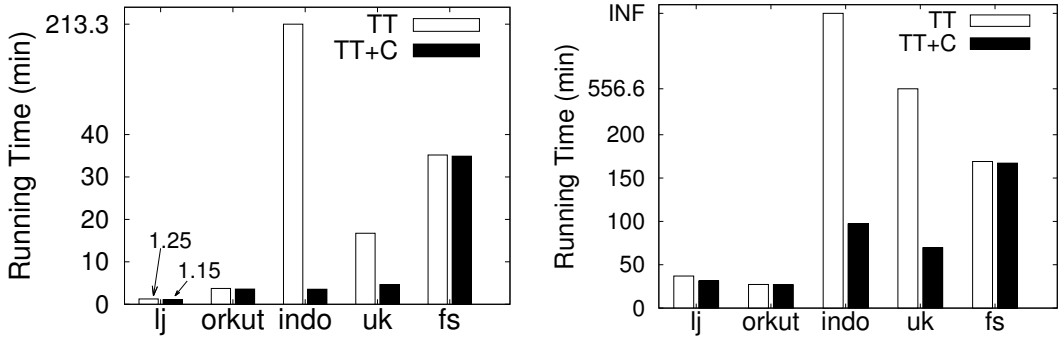
Exp-1: Compressed Graph. In Table 6.3, we observe a more notable compression ratio of the compressed graph built from *eu*, *uk*, *indo* than the other graphs. The reason is, to our best speculation, these three graphs are web graphs, and the web pages from the same domain often reference each other, which tends to forming large cliques. We have found large compressed nodes in the form of cliques in *eu*, *uk* and *indo* of the size 992, 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

Table 6.4: 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	99.92	99.15	99.99	99.97	100	100	100	100
orkut	99.99	99.99	100	100	100	100	100	100
indo	98.13	53.78	99.36	56.65	99.99	99.47	100	100
uk	98.40	70.39	99.58	78.94	99.99	99.99	100	100

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 6.4, 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_2

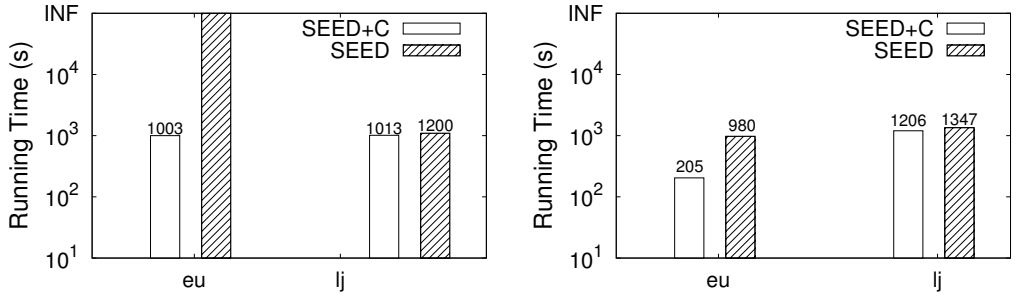
Figure 6.4: The results of Exp-1: TT vs. TT+C.

We next compared the running time of enumerating q_1 and q_2 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 Figure 6.4. The performances are all improved

Table 6.5: Comparison of the size of the output data (in billions) while enumerating q_1 and q_2 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_2	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

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 6.5 for the datasets *lj*, *uk* and their compressed graphs *lj-h* and *uk-h*, while enumerating q_1 and q_2. We obtain a reduction of the output data as expected. The algorithm produces 13% less data on the compressed graph of *lj* in the enumeration of q_1 and q_2, while on *uk-h*, given a higher compression ratio as presented in Table 6.3, it produces 74.4% and 88.80% less data while enumerating q_1 and q_2, respectively.



(a) q_3 (b) q_4

Figure 6.5: The results of Exp-2: SEED vs. SEED+C.

Exp-2: Clique compression. We tested the effectiveness of clique compression

by comparing SEED to SEED+C. We queried q_3 and q_4 on the datasets *eu* and *lj* using both algorithms and present the results in Figure 6.5(a) and Figure 6.5(b). Observe that SEED+C runs faster than SEED in all tests, especially while processing q_3 on *eu*, where SEED+C terminates in 1003 seconds but SEED runs out of time. Note that the effect of clique compression is more notable on *eu* than that on *lj*. The reason is that, to our best speculation, in a web graph like *eu*, web pages within a domain tend to link each other to form large cliques, while in a social network like *lj*, such a strong tie is rarely formed; Obviously, larger clique in the data graph contributes to better clique compression. Although we spend time enumerating and maintaining the large cliques (see $T(C)$ in Table 6.3) for clique compression, the technique does improve the performance of SEED, and it will play an important role when the data graph contains many large cliques (e.g. while processing q_6 on *eu*). This experiment has demonstrated that the performance of subgraph enumeration is further improved after applying clique compression.

6.4 Chapter Conclusion

We introduced two data compression techniques to further improve the performance of subgraph enumeration. The first technique utilizes the properties of the data graph. By aggregating the data nodes that share the same neighbors into a compressed node, we could construct a compressed graph, which improves subgraph enumeration via saving both the computation and communication cost related to the compressed node. The second technique is based on the fact that a large clique in the data graph can represent a large number of matches of a small clique. We hence precomputed the cliques in the data graph, and used them to compress the matches of the clique when it is the join unit in the execution plan.

Chapter 7

Conclusion

Subgraph enumeration is one of the most fundamental problems in graph database with a variety of applications. However, all existing solutions can not scale to web-scale real graphs due to the computational hardness of the problem. In this work, we proposed to solve subgraph enumeration on existing big data processing engine such as MapReduce. We introduced a general decomposition-and-join approach for all proposed algorithms. Based on the general approach, we first proposed the star-based join framework, which used the simple graph storage mechanism that only supports star as the join unit, and applied a left-deep join structure. Based on a well-proposed cost model, we proved that it was sufficient to guarantee instance optimality in the star-based join framework by using TwinTwig, rather than a general star, as the join unit, which inspires the TwinTwigJoin algorithm. TwinTwigJoin can only guarantee optimality in the star-based join framework, with the restraints of the simple graph storage mechanism and the left-deep join structure. Motivated by this, we further proposed the SEED algorithm on the graph-based join framework that guarantees the optimality without the constraints of TwinTwigJoin. In short, SEED implements the SCP graph storage mechanism that supports clique,

in addition to star, as the join units. In addition, a dynamic-programming algorithm was proposed to compute the optimal bushy join plan for SEED. We conducted extensive performance studies to show that TwinTwigJoin is already more efficient and scalable than the state-of-the-art works, while SEED further outperforms TwinTwigJoin by up to two orders of magnitude. Though we have arrived at optimality, the algorithm still suffers from maintaining and transferring the enormous intermediate results (communication cost) while processing complex queries on large data graphs. Aiming at further reducing the communication cost, we proposed two data compression techniques. The first technique transforms the data graph into a compressed graph, and the query processing on the compressed graph can be greatly boosted considering that the huge cost within the compressed nodes has been saved. The second technique utilizes the fact that we can use a large clique in the data graph to represent a large number of matches of a smaller clique when it is the join unit, and hence the results can be expressed in a much more compact way.

Subgraph enumeration also sparks many interesting perspectives that can be considered as future research topics. These include: (1) the distributed join optimization. Now that there are distributed data-flow systems that can process data in a very large scale. There still lack in efficient and scalable techniques that can solve join operation involving many relations in the distributed context. Traditional database heavily relies on indexing and filtering techniques to boost the join. Nevertheless, it is very challenging to adapt these techniques to the distributed system. Essentially, the join operation can be depicted as a graph pattern matching (subgraph enumeration) problem so that we can apply the exisiting graph algorithms, sequential or parallel, to solve join more efficiently. (2) parallel external algorithm. Both IO and computation intensive, it is very challenging to solve subgraph enumer-

ation in scale. To achieve efficiency, researchers are seeking distributed solutions, however, in the reflection of this work, it is extremely challenging to handle that massive data exchange among different machines. A parallel external algorithm may potentially fit into this problem. The external technique can relieve memory burden caused by the massive results, while the multi-core driver can parallelize the execution to a promising extent. Last but not the least, the huge data exchange now happens within one machine instead of among multiple machines that can reside in different geological locations.

Bibliography

[ACL00] William Aiello, Fan Chung, and Linyuan Lu. A random graph model for massive graphs. In *Proc. of STOC '00*, 2000.

[ADH+08] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and Süleyman Cenk Sahinalp. Biomolecular network motif counting and discovery by color coding. In *Proc. of ISMB'08*, 2008.

[AFU13] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating subgraph instances using map-reduce. In *Proc. of ICDE'13*, 2013.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[CLV03a] Fan Chung, Linyuan Lu, and Van Vu. Eigenvalues of random power law graphs. *Annals of Combinatorics*, 7(1):21–33, 2003.

[CLV03b] Fan R. K. Chung, Linyuan Lu, and Van H. Vu. The spectra of random graphs with given expected degrees. *Internet Mathematics*, 1(3), 2003.

[CN85] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1), 1985.

[CSN09] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, November 2009.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI'04*, 2004.

[ER60] Paul Erdos and Alfred Renyi. On the evolution of random graphs. In *Publ. Math. Inst. Hungary. Acad. Sci.*, 1960.

[FFF14] Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. Counting small cliques in mapreduce. *CoRR*, abs/1403.0734, 2014.

[GK07] Joshua A. Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proc. of RECOMB'07*, 2007.

[GRS10] Mira Gonen, Dana Ron, and Yuval Shavitt. Counting stars and other small subgraphs in sublinear time. In *Proc. of SODA'10*, 2010.

[HLL13] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.

[HS08] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proc. of SIGMOD'08*, 2008.

[HTC+] Daniel Halperin, Victor Teixeira, Lee Lee Choo, Shumo Chu, and et al. Demonstration of the myria big data management service. In *SIGMOD '14*.

[IBY+] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07*.

[IK91] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD'91*, pages 168–177, 1991.

[JK84] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, June 1984.

[KWL12] Sanjay Ram Kairam, Dan J. Wang, and Jure Leskovec. The life and death of online groups: Predicting group growth and longevity. In *Proc. of WSDM'12*, 2012.

[LHKL12] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2012.

[LQLC15] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10), 2015.

[LSK06] Jure Leskovec, Ajit Singh, and Jon Kleinberg. Patterns of influence in a recommendation network. In *Proc. of PAKDD'06*, 2006.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. of SIGMOD'10*, 2010.

[MCHW12] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In *WWW*, 2012.

[MMJ+] Zaharia Matei, Chowdhury Mosharaf, Franklin Michael J., Shenker Scott, and Stoica Ion. Spark: Cluster computing with working sets. In *HotCloud'10*, pages 10–10.

[MP08] Tijana Milenkovic and Natasa Przulj. Uncovering biological network function via graphlet degree signatures. *Cancer Inform*, 6, 2008.

[MSOI+02] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594), 2002.

[Pla13] Todd Plantenga. Inexact subgraph isomorphism in mapreduce. *J. Parallel Distrib. Comput.*, 73(2), 2013.

[Prz07] Natasa Przulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2), 2007.

[RR01] Gerta Rücker and Christoph Rücker. Substructure, subgraph, and walk counts as measures of the complexity of graphs and molecules. *Journal of Chemical Information and Computer Sciences*, 41(6), 2001.

[RW15] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc. VLDB Endow.*, 8(5):617–628, January 2015.

[SCC+14] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD'14*, pages 625–636. ACM, 2014.

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. of WWW'11*, 2011.

[SVP+09] N. Shervashidze, S. V. N. Vishwanathan, T. Petri, K. Mehlhorn, and K. M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *AISTATS*, 2009.

[SWW+12] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[TKMF09] Charalampos E. Tsourakakis, U Kang, Gary L. Miller, and Christos Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proc. of KDD'09*, 2009.

[VL05] Fabien Viger and Matthieu Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *COCOON'05*, 2005.

[WC12] Jia Wang and James Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9), 2012.

[WS98] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 6684(393), 1998.

[ZH10] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *PVLDB*, 3(1-2), 2010.

[ZKKM10] Zhao Zhao, Maleq Khan, V. S. Anil Kumar, and Madhav V. Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Proc. of ICPP'10*, 2010.

Appendix A

Appendix

In this appendix, we introduce the algorithm of symmetry breaking in [GK07] that assigns partial order to the pattern graph and formally prove its correctness. we first give some preliminary knowledge of automorphism. Then we show the intuition of eliminating duplicated enumeration by the order-preservation constraint, and present the symmetry-breaking algorithm via assigning the partial orders among nodes in the pattern graph introduced in [GK07]. Finally, we show the correctness of the algorithm.

Automorphism. We denote the automorphism group of a graph Γ as $\mathcal{A}_\Gamma$. We say v_1 and v_2 in a graph Γ is automorphism equivalent, denoted $v_1 \sim v_2$, iff there is an automorphism α of Γ s.t. $\alpha(v_1) = v_2$. The equivalence classes of the nodes of a graph under the action of the automorphisms are called *node orbits*. Here we simply call it orbit. An orbit is trivial if it contains only one node. We denote $\mathcal{O}_\mathcal{A}$ the set of orbits given by the automorphism group $\mathcal{A}$.

We often use a permutation π to express an automorphism, which is further represented in the *disjoint cycle form*.

Example A.1. *Considering an automorphism that maps* $(v_1, v_2, v_3, v_4, v_5, v_6)$

to $(v_3, v_2, v_1, v_6, v_5, v_4)$*, correspondingly. It gives the permutation* $\pi = \begin{pmatrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\ v_3 & v_2 & v_1 & v_6 & v_5 & v_4 \end{pmatrix}$*, which has the disjoint cycle form as* $\pi = (v_1v_3)(v_2)(v_4v_6)(v_5)$.

For the sake of simplicity, we will ignore an element v in the cycle form of an automorphism α if $\alpha(v) = v$. For example, $(v_1v_3)(v_2)(v_4)(v_5)(v_6)$ will be simplified as (v_1v_3). We use $\mathbb{I}$ to denote the identity.

Order-Preservation Constraint and Symmetry Breaking. Suppose a subgraph g of G is isomorphic to the pattern graph P. Let f be the match that maps $V(P)$ to $V(g)$. We consider an orbit $\{v_1, v_2, \cdots, v_k\}$ of P w.r.t $\mathcal{A}_P$, and a data node set $\{u_1, u_2, \cdots, u_k\}$. Without loss of generality, we assume u_1 has the smallest order, and $f(v_i) = u_i$ for all $1 \leq i \leq k$. Taking v_1 as an example, it is clear that there exists an automorphism $\alpha_i \in \mathcal{A}_P$ where $\alpha_i(v_1) = v_i$ for $2 \leq i \leq k$. Therefore, for all the α_i, the mapping $\alpha_i \circ f$ from $V(P)$ to $V(g)$ corresponds to the same subgraph instance g, leading to duplicated enumerations. We then consider an order $<$ among some nodes in the orbit and apply the order-preservation constraint on the mapping (order-preserved mapping). More specifically, if $v_i < v_j$, the mapping f is allowed iff $f(v_i) \prec f(v_j)$. In this case, we enforce the order $v_1 < v_i$ for all $2 \leq i \leq k$, which eliminates all $\alpha \circ f$ where $\alpha(v_1) \neq v_1$ (or similarly preserves only $\alpha \circ f$ where $\alpha(v_1) = v_1$) by order preservation. In this way, we avoid the duplicated enumeration caused by v_1. Note that by enforcing such an order, a mapping f from $V(P)$ to $V(g)$ is valid iff $f(v_1) = u_1$, which means v_1 can only be mapped to a fixed node in a given subgraph instance.

We call the node v in the pattern graph P the *fixed node* if given a subgraph g of G, all the valid matches (according to Definiton 2.2) that map v to a fixed node in g. We know that each node that belongs to a *trivial orbit* is a *fixed node*. As discussed previously, after assigning the orders to v_1, that is $v_1 < v_i$ for all $2 \leq i \leq k$, v_1 becomes a fixed node by the order-preserved matching.

Symmetry Breaking Algorithm. The algorithm in [GK07] first initializes the automorphism group $\mathcal{A}$ as $\mathcal{A} \leftarrow \mathcal{A}_P$. It then runs the following steps iteratively until $\mathcal{A} = \{\mathbb{I}\}$.

- Pick up the largest orbit $\{v_{i_1}, v_{i_2}, v_{i_3}, \cdots, v_{i_k}\}$ from $\mathcal{O}_\mathcal{A}$.
- Assign the order $v_{i_1} < v_{i_2}, v_{i_1} < v_{i_3}, \cdots, v_{i_1} < v_{i_k}$ to make v_{i_1} a *fixed node*.
- Refine $\mathcal{A} \leftarrow \{\alpha | \alpha \in \mathcal{A} \ \wedge \ \alpha(v_{i_1}) = v_{i_1}\}$.

The final step refines the automorphism group $\mathcal{A}$ which contains only the automorphisms that map v_1 to itself. It is easy to verify that after the refinement $\mathcal{A}$ is still a group. Note that after $\mathcal{A}$ is refined, $\mathcal{O}_\mathcal{A}$ is refined correspondingly. We show a running example using the pattern graph presented in Figure A.1.

Example A.2. *The automorphism group $\mathcal{A}$ is initialized as $\mathcal{A}_P = \{\mathbb{I}, (v_1v_2v_3)(v_4v_5v_6), (v_1v_3v_2)(v_4v_6v_5), (v_2v_3)(v_5v_6), (v_1v_3)(v_4v_6), (v_1v_2)(v_4v_5)\}$. The automorphisms partition the nodes into two orbits, namely $\mathcal{O}_\mathcal{A} = \{\{v_1, v_2, v_3\}, \{v_4, v_5, v_6\}\}$. We first pick up v_1 from $\{v_1, v_2, v_3\}$, and assign the order $v_1 < v_2, v_1 < v_3$. We then refine $\mathcal{A}$ to contain the automorphisms that map node v_1 to itself, which gives $\mathcal{A} = \{\mathbb{I}, (v_2v_3)(v_5v_6)\}$, and the new orbits given by $\mathcal{A}$ on the remaining nodes are clearly $\mathcal{O}_\mathcal{A} = \{\{v_2, v_3\}, \{v_5, v_6\}\}$. We further pick up v_2 and assign the order $v_2 < v_3$. After this, $\mathcal{A}$ should be refined to contain the automorphisms that map v_1 to v_1, and v_2 to v_2, which gives $\mathcal{A} = \{\mathbb{I}\}$. The algorithm hence terminates by assigning the following order: $v_1 < v_2, v_1 < v_3, v_2 < v_3$.*

Next, we prove the correctness of the techniques proposed in [GK07] by showing that the results after applying the order restraints given by the algorithm 1) are complete; 2) contain no duplicated results.

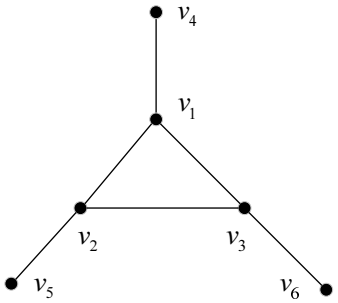


Figure A.1: A pattern graph for symmetry breaking.

Correctness of the Algorithm. 1) Completeness. We prove the completeness of the results by induction on the step of the algorithm. By completeness we mean that the algorithm only eliminates the matches introduced by automorphisms. To start, namely step 0, the results are clearly complete. In step k, we assume that the results are complete. In step $k+1$, according to the algorithm, we pick up an orbit, $\{v_{i_1}, v_{i_2}, \cdots, v_{i_k}\}$ from $\mathcal{O}_{\mathcal{A}}$, and assign the order $v_{i_1} < v_{i_2}, v_{i_1} < v_{i_3}, \cdots, v_{i_1} < v_{i_k}$. We consider $v_{i_1} < v_{i_2}$, and the automorphism $\alpha = (v_{i_1} v_{i_2})$. For any match f where $f(v_{i_1}) \prec f(v_{i_2})$, the order $v_{i_1} < v_{i_2}$ only eliminates $\alpha \circ f$ and nothing more is affected, which means that the order only eliminate duplicated results due to automorphism. We have identical results for the remaining orders. Therefore, the results after assigning the order (by order preservation) are still complete, which completes the proof.

2) No Duplicates. Assume that there are two isomorphisms f_1 and f_2 that map (by order preservation) the pattern graph to the same subgraph in the data graph. There must exist an non-trivial automorphism $\alpha \in \mathcal{A}$ s.t. $f_1 = \alpha \circ f_2$. This is impossible since this contradicts the termination condition of the algorithm in [GK07], where only the identity will be preserved in $\mathcal{A}$.

Given this, we conclude that by assigning the orders via [GK07], we can break the symmetry which ensures, 1) completeness; 2) no duplicates.